2

PROGRAMMING LANGUAGES SERIES

The Design of an Optimizing Compiler

William Wulf
Richard K. Johnsson
Charles B. Weinstock
Steven O. Hobbs
Charles M. Geschke

ELSEVIER COMPUTER SCIENCE LIBRARY

The Design of an Optimizing Compiler

ELSEVIER COMPUTER SCIENCE LIBRARY

Operating and Programming Systems Series

Peter J. Denning, Editor

1. *Halstead* A Laboratory Manual for Compiler and Operating System Implementation

Spirn Program Behavior: Models and Measurement (in prep.)

Programming Languages Series

Thomas E. Cheatham, Editor

1. *Heindel and Roberto* LANG-PAK—An Interactive Language Design System
2. *Wulf et al* The Design of an Optimizing Compiler

Cleaveland and Uzgalis Grammars for Programming Languages (in prep.)

Hecht Global Code Improvement (in prep.)

Maurer The Programmer's Introduction to SNOBOL (in prep.)

Theory of Computation Series

Patrick C. Fisher, Editor

1. *Borodin and Munro* The Computational Complexity of Algebraic and Numeric Problems

Computer Design and Architecture Series

Edward J. McClusky, Editor

Salisbury Microprogrammable Computer Architectures (in prep.)

Artificial Intelligence Series

Nils J. Nilsson, Editor

1. *Sussman* A Computer Model of Skill Acquisition

2

PROGRAMMING LANGUAGES SERIES

The Design of an Optimizing Compiler

William Wulf
Richard K. Johnsson
Charles B. Weinstock
Steven O. Hobbs
Computer Science Department, Carnegie-Mellon University

Charles M. Geschke
Xerox, Palo Alto Research Center.

American Elsevier Publishing Company, Inc.
New York London Amsterdam

AMERICAN ELSEVIER PUBLISHING COMPANY, INC.
52 Vanderbilt Avenue, New York, N.Y. 10017

ELSEVIER PUBLISHING COMPANY
335 Jan Van Galenstraat, P.O. Box 211
Amsterdam, The Netherlands

ISBN 0-444-00158-1 (Paperbound Edition)
ISBN 0-444-00164-6 (Hardbound Edition)

©American Elsevier Publishing Co., Inc. 1975

All rights reserved.
No part of this publication may be reproduced,
stored in a retrieval system, or transmitted
in any form or by any means, electronic,
mechanical, photocopying, recording,
or otherwise, without permission in
writing from the publisher.
American Elsevier Publishing Company, Inc.,
52 Vanderbilt Avenue, New York, N.Y. 10017.

Library of Congress Cataloging in Publication Data

Main entry under title:

The Design of an optimizing compiler.

(Programming languages series; 2) (Elsevier computer science libary)
Bibligraphy: p.
Includes index.
1. Compiling (Electronic computers) I. Wulf, William Allan.
QA76.6.D47 001.6'42 74-21789
ISBN 0-444-00164-6
ISBN 0-444-00158-1 pbk.

Manufactured in the United States of America

We gratefully dedicate this book to the Computer Science Department at CMU, the environment of which is superbly conducive to research, and to the two Als, Newell and Perlis, who were primarily responsible for creating that environment.

Contents

Preface ix

1 Introduction 1
1.1 A Descriptive Notation 3
1.2 An Overview of the Bliss/11 Compiler 6

2 LEXSYNFLO 11
2.1 LEX 11
2.1.1 Lexemes 12
2.1.2 The Symbol Table 12
2.2 A Meta Comment on Switching 15
2.3 SYN 16
2.4 Syntax Errors 20
2.5 FLO 22
2.5.1 Theoretical Background 22
2.5.2 FLO Implementation 34

3 DELAY 45
3.1 Determination of Desirable Feasible Optimizations 48
3.2 Determination of Evaluation Order 50
3.3 Target Paths and Unary Complement Operators 56
3.4 Other Delaying of the Plus Operator 63
3.5 Other Functions of DELAY 68

4 TNBIND 71
4.1 TLA 71
4.1.1 Targeting 72
4.1.2 Cost Determination 77
4.1.3 Lifetime Determination 78
4.1.4 The Runtime Stack 81
4.1.5 Label Assignment 83
4.2 RANK 84
4.3 PACK 85

5 **CODE** 89
5.1 Generating Data Moves 91
5.2 Generating the *incr* Loop Code 97
5.3 Generating Boolean Operations 103

6 **FINAL** 107
6.1 The Final Data Structure 108
6.2 The First Subphase 110
6.3 The Second Subphase 117
6.4 The Third Subphase 118
6.5 An Example 119
6.6 A Final Comment 124

7 **Conclusion** 127

Appendix A Primer on the PDP-11 133

Appendix B A Short Primer on Bliss 141

Appendix C A Complete Example 147

Bibliography 159

Index 160

Preface

This book is concerned with the design and, to a lesser extent, the implementation of an optimizing compiler – a compiler whose paramount goal is the production of extremely efficient object code. Our purpose is to present both new and known techniques in the context of a specific, real compiler and to show how these techniques interact to produce high quality code.

Restricting the presentation to the design of only one compiler permits us to go more deeply into questions mentioned only superficially, if at all, in textbooks on compiler construction. Conversely, these restrictions prevent us from discussing alternate techniques and optimization problems which would arise in compiling for other language/machine combinations. Thus the current work is not a textbook on compiler construction, a survey of optimization techniques, or a presentation of purely original research. Rather, it is an amalgam of all these which attempts to completely and coherently cover the multiple facets and interactions of optimization techniques within one compiler.

The book should be of interest to two classes of computer scientists: those who have an interest in finding out what makes a real optimizing compiler tick, and those who are facing the practical problems involved in actually building such a compiler. Thus the book should be useful in a course on compiler construction as a supplement to one of the standard texts, or as a source book in practical design situations. Of particular importance in both contexts is the fact that we discuss the interactions of the various optimizations; such interactions are not easily treated by articles concerning a single optimization technique or in broader texts.

In presenting the material we have attempted to walk that fine line between a purely formal, implementation-independent treatment and tedious implementation detail. The net effect (we hope) is that: (1) an obvious implementation is implied without ever being explicitly discussed, and (2) the reader is encouraged to devise efficient implementations for the techniques suitable to the language and machine at his disposal.

The book is organized into a set of chapters – each of which describes a phase of the compiler. The chapters are presented in the order in which the

phases are applied to a program segment within the compiler. Although this seems to us to be the only rational organization of the material, the reader should be warned that there is a strong interaction between the various chapters. Thus the reader is well advised to skim the entire book before attempting an in-depth understanding of any one section.

The compiler described in this book translates Bliss/11 [Wul72a] source programs into object programs for the PDP-11 [DEC71]. We recognize that many readers may not be familiar with Bliss, the PDP-11, or both. Appendices have been included on both of these topics. It would be wise to master these before attempting to understand some of the larger examples.

In writing this book we had an ulterior motive – namely to stimulate further research. In the process of designing and building the compiler we uncovered many problems which have received little or no attention from researchers. We found satisfactory solutions for some of these, for others we did not. We suspect that the reasons for the lack of existing solutions arise from two sources: (1) many of the problems arise only in the context of a complete system, and (2) research on a topic tends to breed more research on the same topic – especially if the topic can be expressed in formal terms. Thus by exposing our solutions, both good and bad, and by supplying the context in which these solutions interact, we hope to supply grist for the research mill.

Camera-ready copy for this book was prepared on a Xerox Graphic Printer (XGP) driven by a PDP-11 in the Computer Science Department at Carnegie-Mellon University. The text was formatted on a PDP-10 using PUB, an advanced document production program developed by Larry Tesler and modified for the XGP by Richard Johnsson. Most of the text was originally typed into the computer by Dorothy Josephson and Bunny Kostkas.

Many people have made significant contributions to the design and implementation of the Bliss/11 compiler; we are especially grateful for the contributions of Jerry Apperson, Ron Brender, Paul Knueven, Bruce Leverett, and Dave Wile. Although they are too numerous to mention by name we are also deeply indebted to the users of the compiler who have both made valuable suggestions and have suffered through the compiler's birth pangs. Finally, the compiler was developed under a contract from ARPA (the Advanced Research Projects Agency, Department of Defense) and monitored by the Air Force Office of Scientific Research; we gratefully acknowledge the support, both moral and financial, which allowed the research culminating in this book to be done.

W.W., R.J., C.W., S.H., C.G.

Chapter 1

INTRODUCTION

There are important classes of computer programs which must be highly efficient on a particular computer, independent of how fast that computer may be -- systems programs are one such class. In order to write these programs in a higher-level language and accrue the benefits associated with the use of such languages, we must have compilers which will produce efficient representations of these programs on that particular computer. This book describes one such compiler.

Not all optimization techniques known are employed in this compiler, nor is every aspect original that is discussed. Our purpose is neither to survey known optimization techniques nor to present only original research. Our purpose is, rather, to present various new and known techniques in the context of a specific, real compiler and to show how these techniques interact to produce high quality code.

In describing the compiler in question we have tried to walk that fine line between vague generality and tiresome detail. Where relevant theory exists that theory is presented; where specific algorithms are relevant they are presented in implementation-independent terms; where heuristic policies are used they are clearly labeled as such. We have avoided presenting theory which is not used directly in the compiler. We have also avoided coding details of the implementation. Thus, although we are describing a particular compiler for a specific language and machine, we believe the material to be relevant to other languages and machines.

We hope one effect of this presentation will be to illuminate aspects of compiler optimization which have received little or no attention from researchers. There is an unfortunate tendency for textbooks and journal articles to focus on topics which can be

formalized. Thus a great deal of paper has been devoted to syntax analysis and to those few optimization topics which lend themselves easily to this type of treatment, e.g., to that order of evaluating expressions which minimizes register use. We do not oppose formalization -- it is essential and, indeed, we will use it a good deal. However, the appearance of many papers on a few topics together with a paucity of papers covering complete compilers tends to breed more papers on the same subjects and leave other important problems untouched. We sincerely hope that one effect of this book will be to stimulate research on some of these problems.

Four specific assumptions are made in the remainder of this book:

1. The compiler being described is for Bliss/11 [Wul72a], which is similar to Bliss/10 [Wul71]; a brief introduction to the language is included as Appendix B. The impact of the Bliss/11 language on the compiler strategies arises primarily from three properties of the language:
   a) Bliss does not contain an explicit goto statement [Wul72b]. The absence of a goto obviates the need for certain global flow analysis and permits a more highly recursive and regular structure in the compiler.
   b) Data is typeless. Since they do not apply, certain aspects of more conventional compilers for dealing with type conversion and doing correlated case analysis during code generation will not be discussed.
   c) Any expression may be used as an accessor for a data structure. Therefore all expressions are treated uniformly, i.e., there are no special case optimizations applied to data accessors (e.g., array indices). These optimizations are subsumed into the general strategies of the optimizer.
2. The target machine is the PDP-11, which is a 16-bit minicomputer. Wherever specific examples of the need for or effect of an optimization are needed, the PDP-11 is used. For those unfamiliar with the PDP-11 a brief description of its relevant properties is presented in Appendix A.
3. The problem domain for which Bliss/11 is intended, the

development of production-quality systems programs, warrants the expenditure of a relatively large amount of compile-time in order to achieve runtime efficiency. An explicit goal of the design is to produce better object code than that written by good systems programmers on moderately large programs (say longer than 1000 instructions).

4. Some optimizations involve a speed/size tradeoff. Since the object code is to run on a minicomputer, these tradeoffs have generally been made in favor of the smaller (slower) alternative. Our philosophy is that code size is a more important consideration since we cannot predict which portions of a program will contribute significantly to the execution time, but that space costs something whether or not the code occupying it is ever executed.

The structure of the book is to first present an overview of the Bliss/11 compiler, followed by a more detailed examination of each of its phases. Although the presentation of each of the phases is, to a large degree, able to stand on its own, the reader should realize that there are strong dependencies between them; the presentation cannot totally mask these dependencies. Thus the reader is well advised to skim the entire book before attempting a thorough understanding of any one of the phases of the compiler.

1.1 A Descriptive Notation

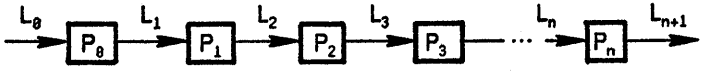
Conceptually a compiler is a transducer which accepts a program

in its "source" language form and produces an equivalent "object" language form. For logical clarity, and sometimes from necessity, a compiler is broken into several "phases." A common diagram which illustrates these phases is

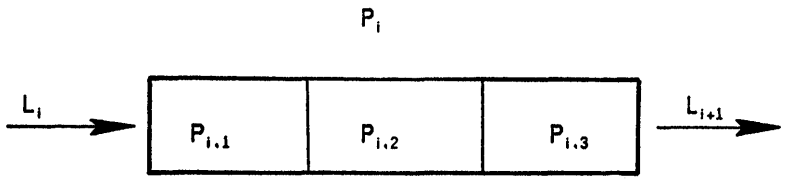
These phases may execute serially (as in overlaid "multiple pass" compilers) or act as coroutines. The diagram above need not distinguish between these modes of operation. In either case some "unit" of the program (anything from a single atom to the entire program) is successively operated upon by each phase. In this view the diagram above is intended only to show data dependencies between the phases--not control flow between them. (Control flow follows, but need not coincide with, the data dependencies.)

Diagrams of the simplicity of that shown above will not be adequate, however. While certain phases must be executed in a prescribed order, others may be performed in arbitrary orders; it may be possible to organize several logical phases into a single physical one. In order to describe these dependencies we have adopted the following notation:

1) A program shall be described as consisting of a linearly ordered set of "phases"; each phase is represented by a rectangular box with a single arrow entering it and another leaving it. Each phase, P_i, is to be thought of as a transducer accepting input language L_i and producing output language L_{i+1}.

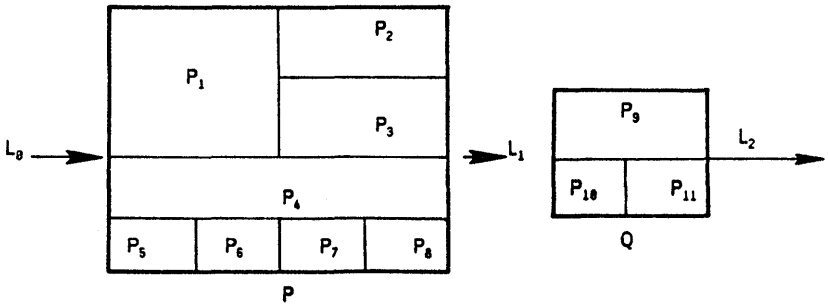


2) The box representing each phase may be divided into a number of vertical "columns." The box, say P_i, then represents a physical phase (operating over an entire unit in L_i) which is divided into logical phases, $P_{i1}, \ldots, P_{in}$, which operate serially on the unit of L_i.



3) Each column may be divided horizontally into "rows" which are subphases of the column which may be executed in an arbitrary order.
4) Rows may be subdivided again into subcolumns, those into subrows, etc. as desired. In each case horizontal separation denotes left-to-right serial dependency, vertical separation (within a column) denotes the absence of such a dependency.

Consider the following example:



This diagram describes a system composed of two physical phases, P and Q. P accepts units of L_0 to produce units of L_1. Similarly Q accepts units of L_1 to produce units of L_2. Within the physical phase P are logical phases $P_1, \ldots, P_8$. From the diagram we see that the sets $\{P_1, P_2, P_3\}$, $\{P_4\}$, and $\{P_5, P_6, P_7, P_8\}$ are independent (may be executed in arbitrary order), but within these sets P_1 must precede both P_2 and P_3, and P_5, P_6, P_7 and P_8 are serially dependent.

1.2 An Overview of the Bliss/11 Compiler

Using this diagrammatic notation, the structure of the Bliss/11 compiler is shown in Figure 1. In the case of Bliss/11, the subroutine is the program unit to which each physical phase is applied. Thus the source text for an entire subroutine is read and the phase LEXSYNFLO applied to it, producing intermediate form L_1. In turn DELAY, TLA,..., and FINAL are applied to the intermediate representations L_1, L_2,..., L_6 for the same subroutine, producing, respectively, L_2, L_3,..., L_7. The next subroutine is processed only after all phases have been applied to its predecessor. A consequence of choosing the subroutine as the unit to which successive phases are applied is that optimizations are applied to this unit; i.e., no optimizations are applied which involve detailed structural knowledge of more than one subroutine simultaneously.

The general attributes of the major phases are summarized below and then explained in more detail in the following chapters.

LEXSYNFLO This phase performs lexical analysis, declaration processing, syntax analysis, and flow analysis. The input is the source program unit in character string form. The output consists of: (1) a set of symbol table entries, (2) a tree representation of the parsed program unit, and (3) a set of lists (generally threads running through the tree) which define feasible global optimizations (constant expressions which may be moved out of loops and the like).

DELAY Delay has three primary functions: (1) to determine the "general shape" of the object code to be generated, (2) to estimate the "cost" of each (linear) program segment, and (3) to determine the evaluation order for expressions. By the "general shape" of the object code, we mean those properties of the operators (e.g., commutativity) or properties of the target machine (e.g., indexing) which may be used to simplify the computation of a value. Decisions are also made at this point as to whether any (or all) of the "feasible" global optimizations are, in fact, desirable. Actual machine

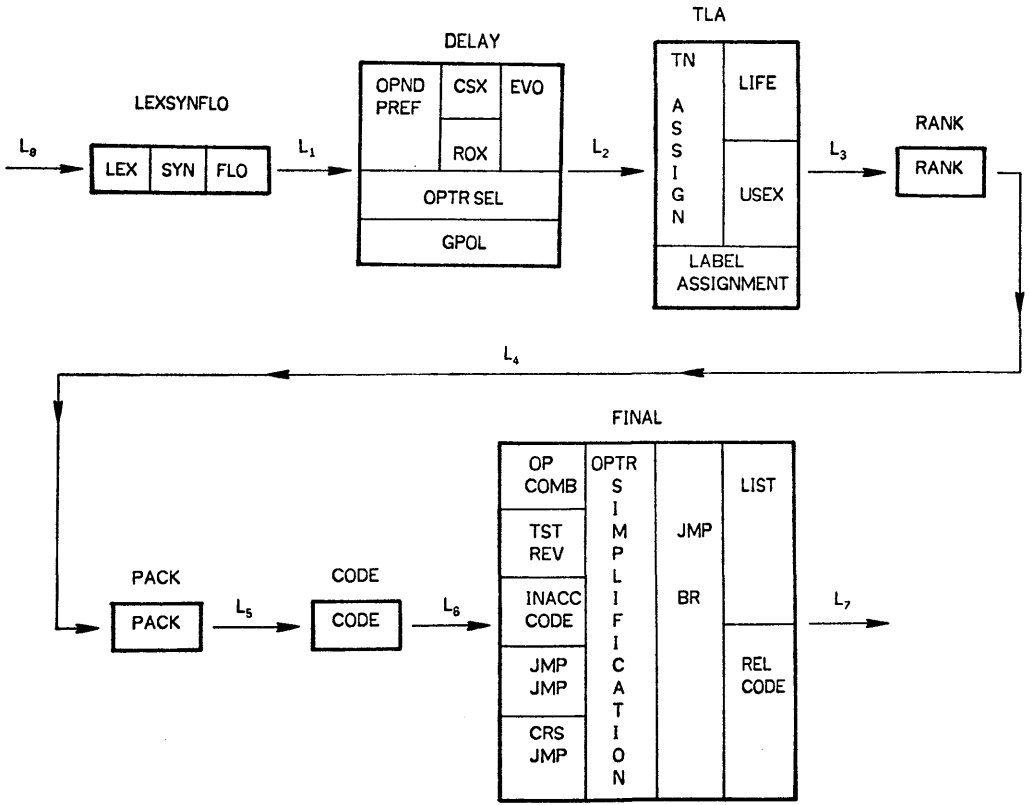


Figure 1. The structure of the Bliss/11 compiler.

code is not generated; rather various flags and fields are set to guide local code generation in a later phase. The cost metric is used to guide selection of evaluation order and in register allocation. The output of this phase is identical to that of LEXSYNFLO (i.e., symbol table, tree, etc.) except that certain information has been added to the tree to signal the subsequent phases of the compiler concerning the shape, cost, and execution order of the code to be generated.

TLA, RANK, PACK

The function of these phases is what in other compilers is frequently called "register allocation"; the difference being that not only registers are allocated, but memory locations as well. The entities which are assigned to locations (registers or memory) include both compiler-generated temporaries and user-defined "local" variables. The output of this phase includes that of DELAY plus the bindings.

CODE

The function of the CODE phase is to produce locally optimal code for each tree node; hence its output is a representation of the target machine language (the tree is discarded at this point). In some cases the locally optimal code is completely determined in DELAY; in these cases the action of CODE is trivial. In many cases, however, further analysis is required. For example, it is CODE's responsibility to determine the optimal sequence of shift and mask instructions to move an arbitrary subfield of one word into an arbitrary position of another.

FINAL

FINAL has two responsibilities. The simpler of these is to prepare the final listing and object code files. The more interesting responsibility is a collection of relatively ad hoc "peephole" optimizations. These optimzations are performed by examining the actual code produced by CODE and eliminating inefficiencies which CODE was unable to detect. For example, FINAL will replace a jump

instruction which transfers to another jump by one which transfers directly to the ultimate destination. It will also remove unreachable code, reverse the sense of certain tests, combine some instructions, etc.

It would be ideal if there were complete algorithms to guarantee the production of optimal programs. A few such algorithms are known [Set70, Ges72] and are used in the Bliss/11 compiler; more often, however, such algorithms do not exist and heuristics must be used. The sequence of phases in the compiler has been designed to provide as much global information as possible on which to base the heuristic decisions. Thus, for example, generation of code is postponed until after both the general shape of the code and binding of temporaries are known. Consequently, special properties of the hardware may be exploited within this context. Similarly, binding of temporaries is done only after the execution order has been firmly established and cost estimates obtained on the basis of the general shape of code to be generated; the first of these permits an accurate characterization of the "lifetime" of the temporary and the second permits realistic tradeoffs to be made between allocating competing entities to registers. Again, evaluation order, general shape, and cost decisions are made in a context where "feasible" global optimizations are known.

Chapter 2

LEXSYNFLO

As discussed in Chapter 1, the primary function of LEXSYNFLO is to generate a tree representation of the executable portion of a program unit together with auxiliary information such as symbol table entries and feasible optimizations. It consists of several subphases: (1) LEX which performs lexical analysis, (2) SYN which performs syntactic analysis and declaration processing, (3) KFOLD which performs compile-time arithmetic, logical, and relational operations (sometimes called constant folding), and (4) GFEAS which forms the actual trees and detects feasible global optimizations such as common subexpression elimination, code motion, etc. Although implemented as a set of potentially recursive subroutines, these subphases actually bear a coroutine relation with each other, with the central control residing in SYN. The subphases are discussed separately below.

2.1 LEX

LEX is essentially a finite state machine which converts the input source text into atoms (or lexemes) for use by later phases of the compiler. Lexemes are the smallest units of program used by these later phases, and consist of the internal representations of identifiers, reserved words, special characters, and literal values. LEX supports multiple input streams which may arise from text files named by the source program or from macro and structure invocations.

2.1.1 Lexemes

Each lexeme contains a type field which specifies the kind of atom it represents; in addition, it carries type-dependent information -- for example:

Literal lexemes contain the actual 16-bit literal value. String literals are represented by lexemes containing a pointer to a linked list structure in which the string is stored.

Delimiter lexemes contain a class (declarator, open bracket, operator, etc.) and an index into a table of routines for performing syntax analysis. Operator lexemes also contain a priority which controls association of operators and operands (e.g., this assures that A+B*C is treated as A+(B*C)).

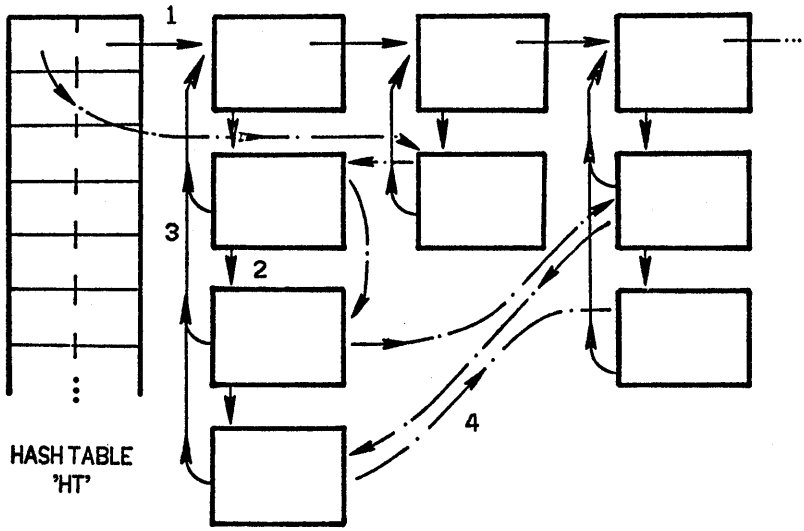
Identifier lexemes contain a pointer to the symbol table entry for the identifier.

2.1.2 The Symbol Table

The symbol table consists of three logically separate tables: the Hash Table (HT), the Name Table (NT), and the Symbol Table (ST). These three tables are linked together along several lists which allow easy and efficient recognition, insertion, and deletion. The relations between these tables, and the linkages between them, are illustrated in Figure 2 and explained further in the following paragraphs.

A hashing function converts the text representation of an identifier to an index into the hash table. Since more than one identifier may hash to the same index, the HT entry serves as the head of a singly linked list of name table entries. Each NT entry contains the actual text representation of the identifier and serves as the head of a singly linked list of all ST entries for that name. The list is maintained so that the ST entry corresponding to the most recently declared instance of the identifier is always pointed to by the NT entry. The HT entry also serves as the head of a singly linked list of ST entries for its hash value. Each ST entry contains a back pointer to its NT entry.

The algorithms for inserting an entry into the symbol table and



1. Name table thread: runs from HT through all NT entries with the same HT index.

2. Symbol stack thread: runs from NT through ST entries with the same name; since this thread is in reverse declaration order, the entry pointed at by a NT entry is always the most recently declared instance of the named entity.

3. Name table backpointer: every symbol table entry contains a backpointer to its name table entry.

4. Symbol table list: this thread runs through all ST entries associated with the same HT index and is in reverse declaration order; it is used to remove ST entries at the end of a block.

Figure 2. Symbol table structure.

for recognizing existing entries are quite simple, and should be fairly obvious from the structure.

> To find the existing symbol table entry associated with a name, first form the hash function value associated with the name. Scan the name table entries which are threaded together from this hash table entry until a character string match is found between the name and the string stored in the name table. The symbol table entry for the currently defined symbol with that name is pointed to by the NT entry.
>
> To create a new symbol table entry for a name, e.g., when that name is (re)declared, search for an NT entry associated with the name as above; if no NT entry is found, create one and place it on the proper HT list. Create a symbol table (ST) entry, and "push" it onto the (possibly empty) stack of entries associated with the NT entry.

Deletion of entries from the symbol table is less intuitive than the insertion and recognition algorithm above, although easily implemented. At the end of a block, all ST entries created in that block must be removed from the symbol table. Since each ST entry contains the block level at which it was created, purging the symbol table is simply a matter of deleting all ST entries whose declaration level is equal to the current block level. These entries are easy to find because the most recent ST entries are at the tops of the ST lists pointed to by the hash table. The algorithm is:

> Scan down the hash table, and at each entry delete the ST entries at the top of the ST list until the declaration level of the top item is less than the current block level, or until the list is empty.

In reality, symbols must only be removed from the ST at the end of a block in the sense that we want to prevent re-recognition. The entry itself continues to exist as long as there are references to it. To this end, symbols removed from the ST are linked onto a list of "purged" symbols. The symbols on this list are actually deleted after code has been generated for the current subroutine.

While it would be possible to process an entire program from source to lexemes in a separate pass independent of the rest of the compiler, this is neither necessary nor desirable. The syntax analyzer operates having knowledge only of a small, current "window" into the program. This window is maintained by LEX and consists of two lexemes -- a "current symbol" and a "current delimiter." These are stored in global variables SYM and DEL.

LEX interfaces with syntax analysis through the routine RUND (Read Until Next Delimiter). RUND is called to update the window by placing the next symbol lexeme (if any) in SYM and the next delimiter lexeme in DEL. The syntax of Bliss guarantees that there is at least one delimiter between each pair of symbols (i.e., Bliss is an "operator grammar" [Flo63]); however, two or more delimiters may appear without an intervening symbol. Thus RUND always fills DEL, but it may place a special "empty" lexeme in SYM.

2.2 A Meta Comment on Switching

In several of the phases to be described below (SYNTAX, DELAY, TNBIND, and CODE), a left-to-right depth-first tree traversal is done. In each of these traversals the actions to be performed depend upon the type of node. A uniform mechanism is used in each phase to accomplish the necessary switching between these actions.

Each type of tree node is uniquely associated with some delimiter in the source language (e.g., the node representing a conditional expression is associated with the <u>if</u> delimiter). Each delimiter lexeme contains a unique, small integer; this integer is stored in the node to identify the node type. Each phase of the compiler contains a vector of the addresses of node-specific action routines for that phase. Switching to the appropriate routine is accomplished by performing a subroutine call indirectly through this vector.*

---

*For those familiar with Bliss, the following is an example of how this switching is actually done in the syntax phase:

<u>bind</u> synplit = <u>plit</u> (soperator, scompound, sif, %...etc. for all syntax routines%);

<u>macro</u> xctsyntax = (.synplit[.DEL])( ) $;

Since traversals and node-specific actions are handled similarly in each phase, the existence of this mechanism will be assumed in each of the discussions below.

2.3 SYN

The compiler uses the top-down method of syntax analysis known as recursive descent [Gri71]. Recursive descent involves a separate, potentially recursive routine to parse each distinct construct in the language. Each routine knows the format of its associated construct and is essentially a simple finite state automaton which invokes other routines to parse subcomponents of itself. The routines are potentially recursive, since embedding in the BNF definition of the syntax admits situations in which a construct may be a subcomponent of itself.

The syntax of the language was designed to simplify the construction of the syntax analyzer. In particular, the following properties are relevant:

1) Every construct is uniquely identifiable by a key delimiter lexeme (e.g., whenever an <u>if</u> lexeme is encountered, an <u>if</u> expression follows).*
2) It is possible to parse the language without backup.
3) The language does not contain the left recursion problem.
4) Bliss/11 is an expression language. In particular this means that all switching between the syntax routines can be done in a single, central routine. Whenever the syntax routine for a particular construct expects a subexpression, it calls this central routine to parse it. This routine makes use of the key delimiter lexemes mentioned above to do the switching between routines.
5) Bliss/11 is an operator language (at least one delimiter will

---

Each occurrence of the macro "xctsyntax" corresponds to a call on whichever syntax routine handles the construct typified by the delimiter lexeme in DEL.

* Note that some meta-variables are used in two different ways (<u>do</u>, <u>while</u>, <u>until</u>). LEX can always determine from context which use is appropriate, and returns the proper version of the lexeme.

appear between any two operands -- e.g., literals or symbols), and requires at most two symbols of "look ahead." As a consequence the syntax analyzer need only "see" at most two lexemes of the input stream at once. These lexemes are the window described in Section 2.1 above.

The task of the syntax analyzer is to transform the infix notation of the input stream into a n-ary tree. We treat only the key meta-variables as operators. Thus there is an <u>if</u> operator whose operands are a boolean part, a <u>then</u> part, and an <u>else</u> part.

Building a recognizer for Bliss/11 is a fairly easy task. A control routine, EXPRESSION, performs the switching; all that the construct associated routines need do is make repeated calls on EXPRESSION, checking for the occurrence of necessary delimiters. For example, the syntax of an <u>if</u> statement is

<u>if</u> e_1 <u>then</u> e_2 <u>else</u> e_3

The recognizer for <u>if</u> first calls EXPRESSION to parse e_1. It then checks for the presence of <u>then</u>, and if it is not there, it indicates an error. If it is present, it then calls EXPRESSION to parse e_2. Finally, it checks for an <u>else</u> and if it is found, it calls EXPRESSION to parse e_3.

Figure 3 illustrates two language constructs and recognizer routines for them. Note that the two routines have similar formats; they initialize error pointers, do the actions specific to the construct, and finally set an error pointer indicating the last expression completely parsed. In fact, each of the several routines needed for a complete recognizer for Bliss/11 would have this same general format. It is this highly regular structure, coupled with the flexibility afforded by algorithmic description of the parse, which makes recursive descent the technique of choice.

To construct a tree we need more than a recognizer: we need a translator. To achieve this, the recognizer routines are augmented with an auxiliary stack (referred to simply as "the stack" for the rest of this section). The purpose of this stack is to hold roots of the subtrees of an expression which is being parsed. Upon entry to a construct-associated routine, the

```
routine sif=
   begin
   ! recognize if e0 then e1 else e2
   init;        ! initialize error pointers
   ruex;        ! parse e0*
   if .del neq "then"
      then return error( );
   ruex;        ! parse e1
   if .del eql "else"
      then ruex;! parse e2 if else present
   end;

routine swhile=
   begin
   ! recognize while e0 do e1
   init;        ! initialize error pointers
   ruex;        ! parse e0
   if .del neq "do"
      then return error( );
   ruex;        ! parse e1
   end;
```

*ruex is an abbreviation for the expression (rund( ); expression( )).

Figure 3. Recognizer routines for if and while.

---

current top of stack is marked so that this construct's intermediate results do not get confused with those results developed at an outer level of the parse. At the end of a construct-associated routine, the contents of this stack (from the mark to the top) are used to create a tree node, and the contents of SYM are replaced by a special lexeme for this subtree.

Specifically, the translator for an if statement calls EXPRESSION to parse e_1. When EXPRESSION returns, a lexeme pointing to the subtree for e_1 is in SYM which is then pushed on the stack (this lexeme is called a graph table lexeme). Then the translator checks for the presence of the then (reporting an error if it is not found), calls EXPRESSION to parse e_2 and pushes a lexeme representing that result onto the stack. If the else is present, EXPRESSION is asked to parse e_3, and again the result is put on the stack. If else is absent, a literal zero is put on the stack instead. Finally, a routine called MAKGT is called to take

the contents of the stack and make a tree node out of them. When it returns, the lexeme pointing to the node is in SYM, and the stack has been popped to remove the subexpression lexemes.

Figure 4 illustrates the translator routines for the constructs that were described in Figure 3. Note the similarity of their formats.

MAKGT has another function. This is to eliminate constant expressions. If it is constructing a tree for an arithmetic operator, for example, and finds that all of the subnodes for this expression are constants, it will perform the arithmetic and return a lexeme for the result instead of a tree. This result may take the form of a literal (in a case such as 2+3) or a new symbol (in a case such as A-3). MAKGT also attempts this "constant folding" for control structures. For example, if it is forming a tree for an <u>if</u> expression, and the subtree for the boolean part is an even literal (false), MAKGT will return only the tree for the <u>else</u> part. Similarly a <u>case</u> expression with a constant selector will cause MAKGT to return only the subtree corresponding to the selected component.

Because we are interested in doing global optimizations, the translator becomes a bit more complicated than the description above indicates. A module called FLOWAN which will be described in Section 2.5 recognizes possible global optimizations. The flow analysis routines operate "inside-out" and are invoked by the syntax routines. In particular, one flow analysis routine is optionally called before a call on EXPRESSION and another is optionally called after EXPRESSION returns. Flow analysis routines are also potentially called before and after calls on MAKGT.

Figure 5 illustrates the translator routines with links to the flow analysis module. Note that the basic structure of the routines does not change.

1) Initialize the error pointers and mark the current top of stack.
2) Do the construct specific actions.
3) Make a tree node from the contents of the stack between the mark and the new top of stack, and reset the stack to where it was on entry. Place a pointer to the new node in SYM.

```
routine sif=
    begin
    ! translate if e0 then e1 else e2
    init;                  ! initialize pointers and mark the syntax stack floor
    ruexpush;              ! parse e0 and push the subtree lexeme on the stack*
    if .del neq "then"
        then return error( );
    ruexpush;              ! parse e1 and push the subtree lexeme on the stack
    if .del eql "else"
        then ruexpush      ! parse e$ and push the subtree lexeme on the stack
        else push(zero);   ! if no else, default to literal zero
    sym←makgt("if");       ! make a tree of the if expression
    end;

routine swhile=
    begin
    ! translate while e0 do e1
    init;                          ! initialize error pointers and mark the stack floor
    ruexpush;                      ! parse e0 and put it on the stack
    if .del neq "do"
        then return error( );
    ruexpush;                      ! parse e1 and put it on the stack
    sym←makgt("while-do");         ! make a tree of the while expression
    end;
```

*ruexpush is an abbreviation for the expression (rund( ); expression( ); push(.sym)) Remember, after a call on expression, sym contains a lexeme, pointing to the tree for the expression scanned.

Figure 4. Translator routines for if and while.

2.4 Syntax Errors

In order to help the programmer pinpoint errors, the lexical analysis routines note and record the line number and beginning character position of each lexeme. Individual syntax routines save these values for the opening bracket of the syntactic construct (e.g., begin, if, etc.), and the most recent "intermediate" bracket (e.g., the most recent ";" in a block). When an error is detected a message is printed describing the error together with pointers to the most recently scanned symbol and the opening and intermediate brackets.

```
routine sif=
    begin
    ! translate if e0 then e1 else e2
    init;                       ! initialize . . .
    ruexpush(fif0);             ! parse e0
    if .del neq "then"
        then return error( );
    ruexpush(fif1);             ! parse e1
    if .del eql "else"
        then ruexpush(fif2)     ! parse e2
        else push(zero);        ! if no else, default to literal zero
    fin("if",fif);              ! make a tree of the if expression**
    end;

routine swhile=
    begin
    ! translate while e0 do e1
    init;                       ! initialize . . .
    ruexpush(fwh0);             ! parse e0
    if .del neq "do"
        then return error( );
    ruexpush(fwh1);             ! parse e1
    fin("while-do",fwh);        ! make a tree of the while expression
    end;
```

*ruexpush(fif0) is an abbreviation for the expression (<optionally call a specific flowan routine for preprocessing e_0>; rund( ); expression( ); push(.sym); <optionally call a specific flowan routine for postprocessing e_0>);
**fin("if",fif) is an abbreviation for the expression (<optionally call a specific flowan routine for preprocessing>; sym←makgt("if"); <it finally calls a specific flowan routine for post processing>).

Figure 5. Translator routines for if and while with flow analysis linkages.

Once an error has been detected, a forward scan is made in an attempt to find a meaningful context in which syntax analysis can resume. Generally such a place is following a close bracket -- notably ")", "end", or ";". However, if an open bracket is encountered in the process of making this forward scan, the syntax analyzer recurs to attempt the analysis of the interior of the bracketed construct. After the interior of the construct has been analyzed, the forward scan resumes.

2.5 FLO

Before describing the actual strategies for global flow analysis in the Bliss/11 compiler a semiformal treatment of the approach will be given. The formulation is due to Geschke [Ges72] and the actual implementation, except for representation issues, models the formulation quite closely.

2.5.1 Theoretical Background

We begin by considering the ordering relations inherent in a representation of a program P. There are several: the lexical order of the input text, the precedence-induced order of evaluation, both data-sensitive and data-insensitive order induced by control flow, and so forth. Two such orderings are important to the development.

The first is the order that results from considering a program as a mapping from its set of input variables to its set of output variables. This ordering, called the <u>essential ordering</u> and symbolized by "$\prec$", is the ordering on evaluation of expressions that results from the application of the data flow and control flow semantics of a language L to the set of expressions E in a program P. The optimizations to be considered will regard the essential order as immutable.

The second ordering allows the selection of subsets, of the total set of expressions in a program, which at a given point are of interest to an optimization strategy. A representation of a program defines (at least partially) an evaluation order on its set of expressions. The ordering inherent in this particular representation may or may not correspond to the $\prec$-ordering.

The <u>initial ordering</u> on a program is symbolized by "$\lhd$". Intuitively, the relation $e \lhd e'$ expresses the notion that in a straightforward evaluation (i.e., that performed by a classical one-pass, non-optimizing compiler) of this representation of the program, the evaluation of e would necessarily have preceded the evaluation of e'. Alternatively, the $\lhd$-ordering captures the notion of a linear flow of control passing through the related expressions. This ordering reflects the precedence relationships of the program; it also reflects the sequential nature of execution as in the case of a compound expression. It does not, on the other hand, necessarily reflect the subnode relationship between

nodes. Below we give a precise definition of the initial order for Bliss; other languages would require a different set of definitions.

Definition

The initial ordering on the set of expressions E of a Bliss program is defined as follows:

Let e be a well-formed Bliss expression. Define $S(e) = \{e' \in E: e' \lhd e$ and e' is a subexpression of $e\} \cup \{e\}$. One of the following cases applies for e:

(1) e_1 <binop> e_2: $e_1 \lhd e,\ e_2 \lhd e,\ e_1 \lhd e_2$
(2) <unop> e_1: $e_1 \lhd e$
(3) begin e_1;...;e_n end: $e_i \lhd S(e_{i+1})\ (1 \le i < n),\ e_n \lhd e$
(4) case e_0 of set e_1; ... ; e_n tes:
$e_0 \lhd e,\ e_0 \lhd S(e_i)\ (1 \le i \le n)$
(5) if e_0 then e_1 else e_2: $e_0 \lhd S(e_1), e_0 \lhd S(e_2), e_0 \lhd e$
(6) select e_0 of nset e_1:e_2; ... ; e_{2n-1}:e_{2n} tesn:
$e_0 \lhd e,\ e_{2i-1} \lhd S(e_{2i})\ (1 \le i \le n)$,
$e_0 \lhd S(e_{2i-1})\ (1 \le i \le n)$
(7) while e_1 do e_2: $e_1 \lhd S(e_2)$
(8) do e_1 while e_2: $e_1 \lhd S(e_2)$
(9) incr 1 from e_1 to e_2 by e_3 do e_4:
$e_1 \lhd e_2 \lhd e_3 \lhd e,\ e_1 \lhd S(e_2),\ e_2 \lhd S(e_3)$
(10) $e_0(e_1, \ldots, e_n)$: $e_i \lhd S(e_i{+}_1)\ (0 \le i < n),\ e_n \lhd e$
(11) leave <label> with e_1: $e_1 \lhd e$.

Then e initially precedes e' (notation: $e \lhd e'$) if and only if in the $\lhd$-transitive closure of E there is a subset $\{e_1, \ldots, e_k\}$ such that $e \lhd e_1 \lhd \ldots \lhd e_k \lhd e'$.

In the case of simple non-control expressions the $\lhd$-ordering reflects the precedence-induced ordering of the binary operation. Most languages, however, contain control environments whose components are potentially $\prec$-order independent. Consider the following compound expression:

$$(A \leftarrow .B;\ C \leftarrow .D;\ E \leftarrow .F)$$

where A, ..., F are distinct variables. Certainly the $\lhd$-order of execution of these three assignments can be altered without violating the $\prec$-order. For example

$$(E \leftarrow .F;\ A \leftarrow .B;\ C \leftarrow .D)$$

produces the same effect.

Even within the context of a simple expression such as

$$.A * .B + .C * .D$$

the $\prec$-order of operations may not be totally defined. For example, Bliss semantics adopt the position that mathematical commutivity implies computational commutivity. Hence, for Bliss expressions of this form, the $\prec$-order of the two products is not defined, while in the initial order: $(.A * .B) \lhd (.C * .D)$. Both ordering relations, however, do reflect the precedence-induced order which requires that the products be evaluated before the addition.

A few additional comments are in order for some of the control expressions in the definition of the $\lhd$-ordering. First, consider the expression

$$e\text{: } \underline{\text{if}}\ e_0\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2$$

Note that e_1 and e_2 are not $\lhd$-related since within the context of the control expression e no control path passes from e_1 to e_2 or vice versa. Notice also that $e_1 \not\lhd e$ since (if e_0 is not constant) there exist environments such that e_1 is not evaluated in the evaluation of e. Similarly $e_2 \not\lhd e$.

The case of a looping expression is slightly more subtle. Consider the expression

$$e\text{: } \underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2$$

and notice that in the definiton of the initial ordering $e_1 \not\lhd e$ and $e_2 \not\lhd e$. The motivation for $e_2 \not\lhd e$ is similar to the <u>if</u>-<u>then</u>-<u>else</u> case. That is, there are environments in which e_1 is false upon entry to e so that control never passes to e_2 within a particular evaluation of e. The motivation for requiring that $e_1 \not\lhd e$ arises from the fact of the cyclic control flow in e. In particular, the subsequent use of the $\lhd$-ordering in defining subsets of a program's expressions identifies those subexpressions of an expression through which a

linear (and a-cyclic) flow of control passes. In summation, $e \lhd e'$ if within the context of the expressions containing e and e' there is an a-cyclic flow of control which passes from e to e' for every environment.

Our observations to this point on the $\lhd$-order and $\prec$-order can be summarized by noting that in general the $\lhd$-order is weaker than the $\prec$-order, i.e., $e \prec e'$ implies that $e \lhd e'$ whereas the converse does not necessarily hold. In some instances the fact that e has been placed "before" e' (in the $\lhd$ sense) by the programmer is essential and sometimes it is not. The optimization strategies discussed below will alter the $\lhd$-ordering in a program. Since the validity of such an alteration is constrained by the $\prec$-ordering, a means must be provided for expressing the validity of transformations of a program. Given a pair of expressions e, e' where $e \lhd e'$, two aspects of the essential ordering can be identified that decide the validity of an optimizing transformation re-ordering e and e'.

<u>Definition</u>

Let $e_1, e_2 \in E$. e_1 is a <u>necessary</u> <u>constituent</u> of e_2 (notation: $e_1 < e_2$) if and only if (iff)

(1) e_1 is a subexpression of e_2, and

(2) evaluation of e_2 requires prior evaluation of e_1.

At first sight conditions (1) and (2) above may appear redundant. Indeed, if the language is Algol, they are redundant. However, in an expression language such as Bliss they are not (e.g. compound expressions).

<u>Definition</u>

Let $e_1, e_2 \in E$. The expression e_1 is an <u>essential</u> <u>predecessor</u> of e_2 (notation: $e_1 \ll e_2$) iff

(1) $e_1 \lhd e_2$.

(2) There exists a context such that the evaluation of the sequences $\{e_1,e_2\}$ and $\{e_2\}$ ($\{e_2,e_1\}$ and $\{e_1\}$) may produce distinct values for e_2 (e_1).

It is important to understand the relationship between the orderings $<$ and $\ll$ and the $\prec$-ordering. If these orderings are

considered in their standard mathematical representations as subsets of ExE, then their relationship can be stated as: $\{\prec\} \subset \{<\} \cup \{\ll\}$. Hence it follows that if $e < e'$ or $e \ll e'$ then $e \prec e'$; or equivalently if e does not precede e' in the $\prec$-ordering then $e \not\prec e'$ and $e \not\ll e'$.

Definition

Let $e_1, e_2 \in E$. e_1 is independent of e_2 (notation: $e_1 \diamond e_2$) iff $e_1 \not\prec e_2$, $e_2 \not\prec e_1$, $e_1 \not\ll e_2$, $e_2 \not\ll e_1$.

Independent expressions are those whose $\prec$-ordering is not determined by the semantics of the language. The usefulness of these primitive relations will become apparent during the discussion of the optimization strategies involving code motions.

Next we introduce an equivalence relation called congruence on ExE, which is an extension of the equality relation on E. Intuitively, two expressions are congruent if there exists a one-to-one correspondence between them that preserves the tree structure and in which the corresponding nodes are identical operators or terminals. More precisely, the elements of E, considered as nodes in the tree representation, can be decomposed into non-terminal (N) and terminal nodes (T). Moreover T itself can be decomposed into names and literals. Hence congruence is defined as follows:

Definition

Let $e, e' \in E$. e is congruent to e' (notation: $e \cong e'$) iff either

(1) $e, e' \in N$,
  e[operator] = e'[operator],
  e[# of operands] = e'[# of operands] = n, and
  $e[operand_i] \cong e'[operand_i]$ $(1 \le i \le n)$;

(2) $e, e' \in T$,
  e and e' are equal literals or identical names.*

The notion of common subexpression (*cse*), which specifies a

* The statement that e and e' are identical names is stronger than character string equality. Here we mean that they in fact refer to the unique variables accessible by that identifier within the present environment.

subset of the collection of all redundant computations, is defined in terms of the primitives developed so far.*

Definition

e and e' are common subexpressions (*cse*'s) (notation: e = e') iff

(1) $e \cong e'$,
(2) $e \lhd e'$ or $e' \lhd e$, and
(3) assuming $e \lhd e'$, $\forall$ e" such that $e \lhd e'' \lhd e'$, $e \not\ll e''$.

The intuition to be conveyed by this definition of a *cse* is that if e = e', then (1) the values returned from the evaluation of e and e' are always identical, and (2) the control flow of P is such that whenever e' is evaluated then e has been evaluated prior to it (or vice versa). The components of the definition mirror this intuition by saying that (1) e and e' are congruent, (2) the evaluation of e initially precedes e' (or vice versa) by definition of the $\lhd$-ordering, and (3) all the expressions that intervene between e and e' have the property that they do not produce side effects that affect the value of e (equivalently: e'), nor does e produce side effects on them. The latter condition says intuitively that the evaluation of e' is unnecessary since its value is available from the evaluation of e.

The literature on object code optimization in the presence of control flow identifies a collection of optimization strategies called code motions. The code motion optimizations about to be described are all predicated on a recursive inside-out approach for their detection. For example, in detecting code motions relative to an if-then-else control environment, the detection proceeds by first invoking the optimization on the then and else expressions. The optimization on each of these expressions will (1) detect the feasible optimizations within its own local

* It should be noted that the concept of common subexpressions as defined here is a subset of the concept as it has been used in some earlier papers. We shall refer to the more global concept as "redundant expressions," which is consistent with yet other papers. The redundant computations not captured by our definition are covered by other notions introduced later.

environment, and (2) return information to be used in detecting optimizations relative to the <u>if-then-else</u> environment. This overall approach requires that a means be provided for stating precisely what information about the subcomponents of a control expression is required in order to detect optimizations for the control expression itself. The notion of a <u>linear block</u>, β, is introduced for this purpose. Roughly speaking, β corresponds to those subexpressions of e through which a linear, a-cyclic (i.e., $\lhd$-order) flow of control passes.

<u>Definition</u>

Let $e \in E$ and $E' = \{e' \in E: e' \text{ is a subexpression of } e\}$. The <u>linear block</u> β relative to e (notation: $\beta|e$) is the set $\beta|e = \{e' \in E': e' \lhd e\}$.

Since in the context of the use of $\beta|e$ the expression e is quite often obvious, "|e" is simply omitted in most cases; otherwise, by convention, the linear block relative to e_i will be denoted by β_i. In flow diagrams, linear blocks are depicted as unbroken vertical lines (flow passing from top to bottom):

$$|\beta$$

The following definition introduces three sets which make the succeeding definition less cumbersome.

<u>Definition</u>

Let $e \in \beta$, β a linear block.

pro-dominator$(\beta,e) = \{e' \in \beta: e' \lhd e, e' \ll e\}$,

epi-dominator$(\beta,e) = \{e' \in \beta: e \lhd e', e \ll e'\}$,

post-dominator$(\beta,e) = \{e' \in \beta: e \lhd e', e' \not\lhd e\}$.

The pro-dominator set contains those elements of β which initially precede e such that they produce a side effect on e or e produces a side effect on them. The epi-dominator set differs from the pro-dominator only in that its elements initially follow e. Intuitively the pro-dominator (epi-dominator) contains those elements of β which prevent the movement of e backward (forward) to the head (tail) of β because they produce a side effect on e or vice versa. The post-dominator set consists of

those elements of β which initially follow e and are not independent of e. Hence the post-dominator consists of those elements which prevent the movement of e forward either because of a side effects relationship or because their evaluation requires the evaluation of e. It follows from the definitions of "$\ll$" and "$\diamond$" that: epi-dominator(β,e) $\subseteq$ post-dominator(β,e).

Definition

Let β be a linear block.

prolog(β) = {e ϵ β: pro-dominator(β,e) = 0},

epilog(β) = {e ϵ β: epi-dominator(β,e) = 0},

postlog(β) = {e ϵ β: post-dominator(β,e) = 0}.

The prolog(β) is that set of expressions which have no pro-dominators in β; thus the elements of prolog(β) represent computations which might safely be moved backward to the head of β, or perhaps beyond, if this were desirable. Similarly the expressions in epilog(β) and postlog(β) have no epi-dominators or post-dominators, respectively, in β. Thus the elements of these sets represent computations which may under certain circumstances be moved forward to the end of the linear block. Note that it follows immediately that postlog(β) $\subseteq$ epilog(β).

Code Motion Optimizations

We can now define various code motion optimizations. Consider, for example, a branching control construct of the form shown in Figure 6 where ϵ functions as a selector among the n branches. This form of control represents both if-then-else and case types of control environments.

The first feasible optimization exploits code motions out of the linear blocks $\beta_1, \ldots, \beta_n$ to produce a flow diagram of the form shown in Figure 7. The linear blocks α and ω contain those expressions factored forward and backward from all of the branches, β_i.

A primary goal of the development is to provide a means of concisely describing the set of feasible members of sets such as α and ω. To this end an operator on the power set of E is introduced.

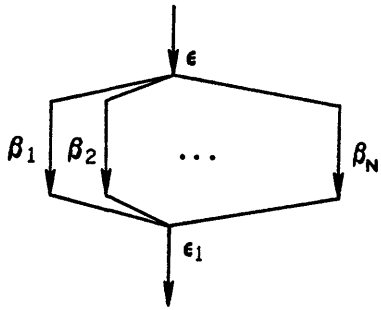


Figure 6. An n-way branching structure.

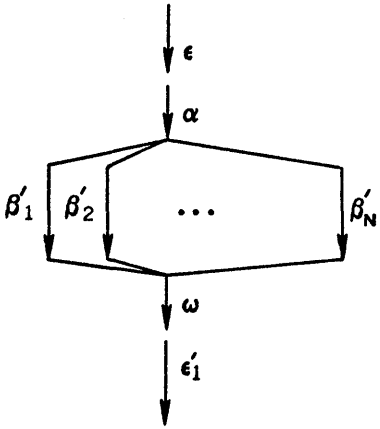


Figure 7. α and ω code motion.

<u>Definition</u>

Let $E_1, \ldots, E_n$ be subsets of E. The <u>formal intersection</u> of the sets E_i is defined as

$$\wedge E_i = \{e \in E\colon \forall\ i,\ 1 \le i \le n,\ \exists\ e_i \in E_i \text{ such that } e \cong e_i\}.$$

While formal intersection is different from ordinary set intersection the analogy should be obvious; it differs only in that the equivalence relation of equality of elements is replaced by that of congruence.

The notion of formal intersection provides us with a powerful tool to concisely define the sets α and ω.

> Given a forked control environment with branches $e_1, \ldots, e_n$, the domain of elements (α) available for pre-evaluation is described by: $\alpha \subseteq \wedge$ prolog(β_i). The domain of elements (ω) available for post-evaluation is described by: $\omega \subseteq \wedge$ postlog(β_i).

Moreover,

> Given a forked control environment with selector expression ϵ and branches $e_i = (\epsilon; e_i)$ and $\beta_i' = \beta_i'|e_i'$, $1 \leq i \leq n$. Then the set of expressions whose evaluation at the merge point would be redundant is the set: $\wedge$ epilog(β_i').

The looping constructs we shall consider consist of a body β_1 and a predicate β_2 to be evaluated on each iteration. We shall consider two types, a "while-do" form which has its test at the top of the loop and a "do-while" with its test at the bottom of the loop. These constructs are illustrated in Figures 8a and 8b respectively. Other forms of loops such as counting types can be modeled by these forms.

The first optimization is the pre-evaluation of the "loop invariant expressions," i.e., those whose values do not change on any iteration of the loop.

> Given a loop control environment, the set of loop invariant expressions is described by: χ = prolog(β) $\cap$ epilog(β), where β is the linear block relative to the compound expression (β_1; β_2) in the "do-while" case and (β_2; β_1) in the "while-do" case.

The description has intuitive appeal since it simply states that any expression whose evaluation is not affected by occurring either before or after the loop is not changed by execution of the loop.

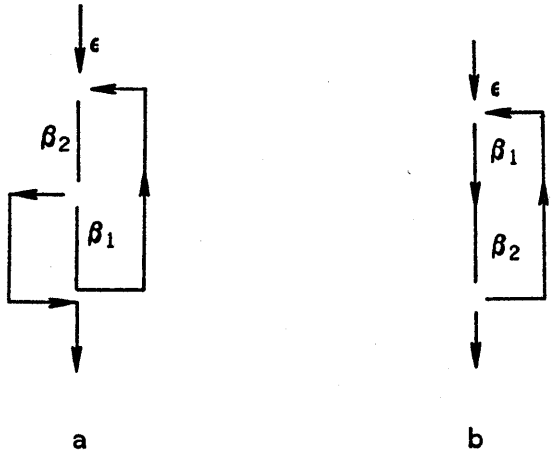


Figure 8. "while-do" and "do-while" constructs.

Cyclic Reevaluations

The cyclic nature of loop control gives rise to a particular class of "redundant" computations. Consider the example shown in Figure 9. Clearly the expression .A*.B is not invariant throughout the loop. However if the expression .A*.B were pre-evaluated at entry to the loop and stored in a temporary T and if after each computation of A or B the expression .A*.B were again evaluated in T, there would be no need to reevaluate .A*.B at the top of the loop on each iteration. The restructured computation is shown in Figure 10.

> Given a loop control environment where β is the linear block relative to the expression $(\beta_1; \beta_2)$ ("do-while") or $(\beta_2; \beta_1)$ ("while-do"), the set of expressions whose evaluations at the head of β are redundant to evaluations at the tail of β are described by the set: $\rho = \text{prolog}(\beta) \wedge \text{epilog}(\beta)$.

Finally, let us point out how loops participate in the exposure of the set of redundant expressions to their surrounding environment.

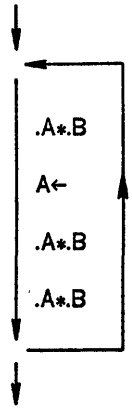


Figure 9. Sample looping construct.

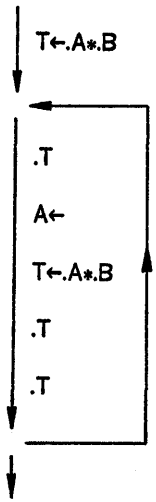


Figure 10. ρ code motion in a loop.

In the case of a "while-do" construct (Figure 8a) the set of expressions whose values are available on exit from the loop is the set epilog(ϵ; β_2).

For the case of a "do-while" construct (Figure 8b) the set of available expressions on exit is epilog (ϵ; β_1; β_2).

2.5.2 FLO Implementation

In the preceding theoretical background the importance of congruent expressions should be obvious. Therefore the structure of FLO has been organized to make the recognition of congruent expressions extremely efficient. Because it is efficient to do so, *cse*'s (in the sense defined above) are also detected at the same time. Figures 11 and 12 illustrate the data structures used for this recognition -- principally a set of threads running through the tree representation of the program.

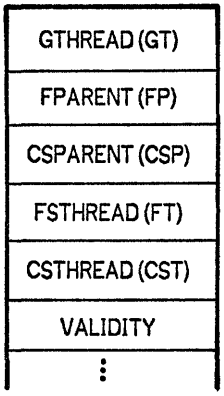


Figure 11. Data fields for FLO.

Suppose the set of expressions in a particular program is denoted by E and that a subset of these, C, is congruent. Each element of E is a node in the tree representation of the program. One element of the set C is designated to act as the representative of the class and is called the "formal parent" of the class. Every element of the class contains a reference to the

formal parent (FPARENT field in Figure 11). Furthermore, the elements of C may be partitioned into one or more sets of *cse*'s, $C_0,\ldots,C_n$, each of which is a unique collection of *cse*'s. An element of each of the sets C_i is designated as the representative of the set and called its "*cse* parent." Elements of C_i all contain a reference to this representative (CSPARENT field in Figure 11). Elements of C_i are threaded together from the representative (through the CSTHREAD field) and CS-parents are threaded together from the formal parent (through the FSTHREAD field). Formal parents for expressions involving the same root operator are threaded together (through GTHREAD, Figure 11); this thread is rooted in an auxiliary vector called GTHASH.

The following discussion is divided into two pieces which are actually merged in the implementation. The first piece deals with the recognition of *cse*'s "on-the-fly," that is, as the program is parsed and the tree representation program is built. The second piece of the discussion deals with recognition of additional redundant expression evaluations which cannot be safely recognized "on-the-fly" and with detection of feasible code motion optimizations.

On-The-Fly Recognition

Congruent expressions and (most) *cse*'s are recognized as the program is parsed and the tree representation is built. Two expressions are congruent iff: (1) they involve the same operator, (2) they involve the same number of operands, and (3) all of their corresponding operands are congruent. Therefore, to determine whether an expression which is just being built is congruent to an existing one the compiler searches the GTHREAD corresponding to the operator of the new expression. Each node on this list is the representative of a distinct equivalence class under congruence. Hence, for each node on the list the compiler tests to see whether the number of operands is identical to that of the new expression and checks whether the corresponding operands are congruent. Note that $e \cong e'$ if and only if e[FPARENT] = e'[FPARENT], so the latter check is simple. If the new expression is not congruent to existing ones, the node is added to the GTHREAD thread as the representative (formal parent) of a new equivalence class; otherwise, a check is made to

determine whether the new expression belongs to one of the *cse* classes.

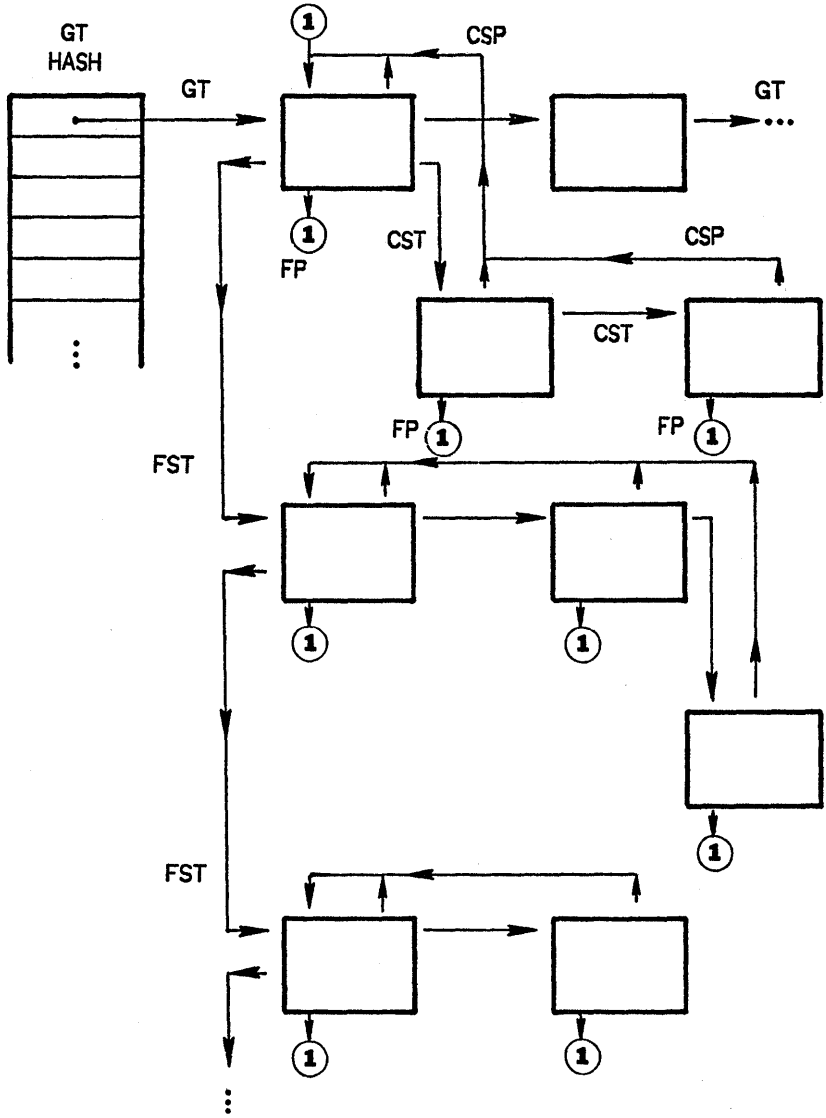


Figure 12. FLO links in the Graph Table.

Expressions e and e' are *cse*'s, e = e', iff: (1) $e \cong e'$, (2) $e \lhd e'$, and (3) $\forall e'' \ni e \lhd e'' \lhd e', e \not\ll e''$. The first condition, congruence, is detected as described above. The second and third condition are encoded in the VALIDITY field (Figure 11) in a manner which will be described below. The general strategy is, having detected that a new expression is congruent to one or more extant ones, to scan from the formal parent through the *cse* parents (via the FSTHREAD) to determine whether there exists one of these whose value is valid at that point. If so, the new expression is added to the appropriate set; otherwise, the expression becomes the *cse* parent of a new set.

The validity field consists of six subfields as shown in Figure 13:

PD	"Purged," one bit, set when node is permanently invalid for *cse* recognition.
RM	"Real mark," one bit, set when node is invalid in the current linear block.
JM	"Join mark," one bit, set when real mark must be set at join of a forked construct.
MM	"Must mark," one bit, set when effect of a side effect must be noted later.
CRLEVEL	"Creation level," encoding of the linear block, β, in which the expression was created.
MKLEVEL	"Mark level," encoding of the outermost linear block, β, in which the expression was marked, i.e., invalidated.

PD	RM	JM	MM	CRLEVEL	MKLEVEL

Figure 13. Subfields of VALIDITY.

These fields are maintained, together with global variables CEILING and FLOOR, such that the value represented by a node is valid if:

$$\sim PD \wedge \sim RM \wedge (CRLEVEL \leq CEILING) \wedge (CRLEVEL \geq FLOOR)$$

The encoding of the validity field is possible because of the fully nested (i.e., no goto) nature of Bliss and the "inside-out" invocation of the recognition mechanism. This mechanism will be explained in the context of the processing of two specific syntactic constructs: the if-then-else and do-while. Before describing the handling of these constructs, however, several aspects of the implementation must be described.

(1) A global variable, LEVEL, is maintained such that its value is incremented whenever a new linear block is encountered and reset when the end of that linear block is detected. Therefore a necessary, but not sufficient,* condition for $e \lhd e'$ is $e[CRLEVEL] \leq e'[CRLEVEL]$. (The variables CEILING and FLOOR assume values of LEVEL and define a "window," $FLOOR \leq L \leq CEILING$, in which "on-the-fly" recognition of *cse*'s is possible. The interpretation of these variables will be discussed in context below.)

(2) Whenever a side effect producing operation, e.g., assignment or subroutine invocation, is encountered, the "must mark" bit of nodes whose value depend upon the affected variables is set. As may be seen from the definition of validity given above, setting this bit does not invalidate the value of the node. The semantics of Bliss specify that the effect of side effects need to be accounted for at prescribed places, notably at the semicolon. Hence at the appropriate points a primitive, "markmmnodes," is invoked to convert MM-bits to RM-bits (at which time the MKLEVEL field is also set).

(3) The construction of the recursive descent parser is such

* The condition is not sufficient since linear blocks at the same nesting level need not satisfy the initial ordering; for example, the then and else portions of the conditional are at the same nesting level, but neither initially precedes the other.

that any invocation of the driver, EXPRESSION, can be (optionally) preceded and/or followed by invocation of a context-specific flow primitive. The particular case of the conditional expression, for example, involves invocation of six flow primitives, f_0-f_5, as shown below.

```
if   e0   then   e1   else   e2
   ↑  ↑        ↑  ↑        ↑  ↑
   f0 f1       f2 f3       f4 f5
```

Ignoring the recognition of the α and ω sets described earlier, the flow primitives for the conditional expression are:

```
f0:
f1:  markmmnodes; increment (ceiling);
f2:
f3:  markmmnodes; refresh;
f4:
f5:  markmmnodes; refresh; decrement (ceiling); join;
```

where

```
markmmnodes:  ∀e (e[rm] ← e[rm] v e[mm]; e[mm] ← 0);
increment(x):  (level ← level + 1; x ← level);
decrement(x):  (level ← level - 1; x ← level);
refresh:       ∀e (if e[CRLEVEL]≥ CEILING
                     then e[PD] ← 1
                     else if e[MKLEVEL]≥ CEILING
                         then
                             begin
                             e[jm] ← e[jm] v e[mm] v e[rm];
                             e[rm] ← e[mm] ← 0
                             end);
join:    ∀e ∋ e[MKLEVEL] ≥ level (e[rm] ← e[rm] v e[jm]);
```

The operation of this collection of routines is probably best explained by tracing its execution for the expression "if e_0 then e_1 else e_2." The boolean expression, e_0, is an element of the linear block determined by the context in which the entire expression occurs. If e_0 contains uses of *cse*'s created, earlier

they will be recognized; also new *cse*'s may be created in e_0. If e_0 produces side effects, the MM bits of affected nodes will be set. After e_0 has been parsed, however, a flow primitive, f_1, is invoked. One of the prescribed places in Bliss at which side effects are to be accounted for is following such a boolean expression; hence "markmmnodes" is invoked to set RM bits. At this point LEVEL and CEILING are incremented to reflect that a new linear block will be entered to evaluate e_1.

During the parse of e_1, *cse*'s will be recognized, potentially new *cse* parents generated, MM and RM bits set, etc. After e_1 has been completely parsed the flow primitive f_3 is invoked and performs "markmmnodes" and "refresh." The purpose of refresh is: (1) to "purge" all nodes created "in the attic," i.e., with a creation level ≥ CEILING, and (2) to set the "join mark," JM, bit of nodes invalidated in the attic (and simultaneously reset their RM and MM bits). During the left-to-right parse of the program we must assume that values created on one branch of the conditional will not be available after the forks rejoin; hence the purge of nodes created in the attic (we shall come back and pick up values created on both branches later). On the other hand, nodes created prior to the branch and invalidated on one branch will be available for the other; hence nodes marked along the branch are reset to "valid" (RM reset to zero). The JM bit of these nodes is set, however, to serve as a reminder that the value of the expression will not be available after the forks rejoin.

The else-branch, e_2, is treated similarly. Notice, however, that neither CEILING nor LEVEL is altered since both branches, even though they are distinct linear blocks, are at the same nesting level. Also, after e_2 is parsed, LEVEL and CEILING are reset and "join" is invoked to set the RM bits correctly for nodes which were invalidated along either (or both) branches.

The <u>do</u>-<u>while</u> expression invokes flow primitives f_6-f_9 as shown below:

$$\underline{\text{do}} \quad \underset{\substack{\uparrow \quad \uparrow \\ f_6 \quad f_7}}{e_0} \quad \underline{\text{while}} \quad \underset{\substack{\uparrow \quad \uparrow \\ f_8 \quad f_9}}{e_1}$$

where

f_6: increment (floor);

f_7: markmmnodes;
f_8:
f_9: markmmnodes; decrement (floor);

Since in the case of a do-while, both e_0 and e_1 are guaranteed to be executed at least once, there is major problem with "on-the-fly" recognition of *cse*'s. We wish to prevent recognition of expressions whose value may be valid on the first execution of the loop but, because of side effects later in the loop, will be invalid on the second and subsequent iterations. The manipulation of FLOOR, together with the definition of "valid" expressions given earlier, prevent recognition of such expressions. Values available prior to the loop and not altered by the loop are detected at a later stage in a manner which will be described below.

Other Redundant Expressions and Code Motion Optimizations

The detection of the remaining redundant expressions and feasible code motion optimizations hinges primarily upon formation of the prolog, epilog, and postlog sets described earlier. To facilitate the formation of these sets a new global variable, the "atomic-block-count," or ABCOUNT, is introduced. This variable monotonically increases whenever side effects are invoked; an auxiliary stack is maintained such that the top element of this stack contains the value of ABCOUNT when the current linear block was entered. Also, each symbol table entry for a variable in a program is augmented with two sets -- CHANGELIST and USELIST. Whenever a value is changed or used, an entry consisting of the atomic block count and a pointer to the node at which the action occurred is made in the appropriate set. The sets themselves are represented by linked lists; the references are ordered in a manner which minimizes the search time for the intersection operators.

The prolog set consists of those expressions which have no essential predecessors in the block. This set is formed "on-the-fly" as the program is parsed. A newly formed node, e, belongs to the prolog of its linear block iff no previous operation in the linear block has had a side effect on the operands of e. There are three interesting cases:

(1) $e = e_1 \langle binop \rangle e_2$:

$$e \in prolog(\beta) \text{ iff } e_1 \in prolog(\beta) \wedge e_2 \in prolog(\beta) .$$

(2) $e = .e_1$: $e \in \text{prolog}(\beta)$ iff

(a) e_1 is a variable and e[changed] < ABCOUNT
$\forall x \in \text{CHANGELIST}(e)$
(x[ABCOUNT] < ABCOUNT(β)),
where ABCOUNT(β) is the top of the auxiliary stack of ABCOUNT values described above,

(b) or e_1 is an expression and $e_1 \in \text{prolog}(\beta)$.

(3) e = <constant>:
by definition, all constants are in prolog(β) .

Although these definitions are recursive, the information necessary to make the first test (1) can be (and is) encoded in a single bit in each node.

The epilog set consists of those expressions in a linear block which are not affected by expressions following them in the block; that is, those expressions are not the essential predecessor of any expression in the block. It follows from the earlier discussion of "on-the-fly" recognition of *cse*'s that

$$e \in \text{epilog}(\beta) \text{ iff } e \in \beta \wedge \text{PD} \wedge \text{RM} .$$

Thus at the end of a linear block a simple linear scan of the expressions is adequate to determine membership in epilog(β).

The postlog set contains those expressions in a linear block which are neither essential predecessors nor necessary constituents of the expressions which initially follow them in the linear block. Since postlog(β) $\subseteq$ epilog(β), it is only necessary to determine those nodes which are not necessary constituents of following nodes; in the case of Bliss the only candidates are non-value-producing components of a compound expression (e.g., e_1, e_2, ..., e_{n-1} in (e_1; ...; e_{n-1}; e_n)) so that a simple tree walk from the root of the linear block will detect the elements of postlog(β).

Returning to the example of the conditional expression

$$\begin{array}{ccccccccc} \underline{\text{if}} & e_0 & \underline{\text{then}} & e_1 & \underline{\text{else}} & e_2 \\ & \uparrow\ \uparrow & & \uparrow\ \uparrow & & \uparrow\ \uparrow \\ & f_0\ f_1 & & f_2\ f_3 & & f_4\ f_5 \end{array}$$

and concentrating only on those operations necessary to detect feasible optimizations not exposed in the previous section, we have

f_0:
f_1:
f_2: pushflow;
f_3: $pr_1 \leftarrow$ prolog; $ep_1 \leftarrow$ epilog; $po_1 \leftarrow$ postlog; popflow;
f_4: pushflow;
f_5: $pr_2 \leftarrow$ prolog; $ep_2 \leftarrow$ epilog; $po_2 \leftarrow$ postlog; popflow;
$\alpha \leftarrow pr_1 \wedge pr_2$; $\omega \leftarrow po_1 \wedge po_2$; $\pi \leftarrow ep_1 \wedge ep_2$;

where: (1) "pushflow" increments ABCOUNT, pushes the stack of these values, and sets up a new prolog set; (2) "popflow" pops the stack of ABCOUNT values, creates the "prolog," "epilog" and "postlog" for the linear block and returns a reference to them; and (3) "$\wedge$" is the formal intersect operator. Note that the set "π" is the set of expressions created on both branches whose values are therefore available after the merge point; expressions whose values are available after the merge point, because they were created prior to the fork point, are handled by the mechanism described previously.

Although we have avoided describing all the details of the implementation, we hope that the description is adequate to suggest the general approach. Before leaving the subject of FLO it should be reemphasized that the only action taken is to detect feasible optimizations. The full tree representation of the program remains. Threads have been added to link together *cse*'s, and separate lists have been built to identify feasible code motions, but no transformations have been performed on the tree itself. Hence later phases, especially DELAY, may or may not choose to exploit these feasible optimizations, depending on context.

Chapter 3

DELAY

As mentioned previously, DELAY has three primary functions: (1) to determine the "general shape" of the ultimate object code to be produced, (2) to form an estimate of the cost of each program segment, and (3) to determine evaluation order for the expressions in a program segment. Although it is impractical to specify how each of these functions is implemented for each syntactic construct, detailed discussions of various important situations will be given.

Before proceeding to details, however, an overview of the structure of DELAY is desirable. All functions (subphases) of DELAY as shown in Figure 1 are performed on a single tree walk. Node-specific action routines are invoked as each node is encountered on the way down the tree. Each such routine tends to have the following structure:

1) When control is received from above, a certain amount of "context information" and a "preferred shape" are passed down. The node-specific routine examines this information to make a preliminary determination about two things: (a) whether or not the feasible optimization for this node, as detected by FLO, is desirable, and (b) the "preferred shape" of the operands of this node.
2) The preliminary decisions made in (1) become the "context information" and "preferred shape" to be passed to the subnodes of this node. Thus the node-specific action routines next invoke the node-specific action routines at the next lower level; these will return information concerning the "actual shape" of the operands when they complete. In some cases the preliminary decisions made in (1) are refined, and the context information to be passed

down to the ith subnode updated, as information about actual shape of previous subnodes becomes available.

3) After the delay operation has been applied to all subnodes of the node in question, the node-specific action decides upon the actual shape of the node itself. It bases this decision upon the "preferred" shape passed from above and the "actual" shape of its operands. At this point the node-specific action routine also determines the evaluation order of the subexpression's operands and estimates the cost of evaluating the node (including its subnodes).

Admittedly the terms "preferred shape," "actual shape," "context information," etc. are vague. We have left them vague at this point because there are many different kinds of information passed up and down the tree, most of which will be explained in detail in the subsequent sections. However, since this general mechanism is crucial to the overall quality of the final code produced, some more specific examples are in order at this point. Some of the kinds of context information passed down the tree are:

The kind of value needed. In certain contexts the value of given expression may not be needed at all, in other cases the result of an expression may be used to generate control flow, and finally in many contexts the value of an expression must be a genuine bit pattern.

The kind of addressability required of the expression. The expression appearing on the left hand side of an assignment operator must specify the address of a ≠location into which a value will be stored. On the other hand, the expression on the right hand side of an assignment operator must evaluate to the value to be stored. Because of asymmetries in most hardware the two contexts are not identical; hence the expression delayers must know which of the two contexts applies.

Sign preference. In many cases it is less expensive, locally, to generate the negative of a result than the value with its correct sign. In some cases this local optimality leads to global optimality and in others it does not, depending on context.

Corresponding to each of these pieces of information passed down the tree there is actual information passed back up, e.g., the actual kind of value generated, how that value may be addressed (if at all), and the actual sign of the result. In most cases the preferences expressed by the higher levels of the tree walk are not binding on the lower level ones. The lower level node-specific routines conform to the preferences expressed by the higher level ones only if they can do so at a cost which is less than that which would be incurred by generating the appropriate transformation at the higher level.

In the subsequent sections of this chapter we consider each of the subphases of DELAY separately. In reading these the reader is strongly urged to keep in mind that the subphases are actually merged in the implementation and that many of them are split into two parts -- namely, those actions which make a preliminary determination of the context information for subnodes and thus occur before delaying them, and those actions which determine the actual shape of the given node and are performed only after the actual shape of the subnodes is known.

In terms of the notation of Figure 1, GPOL is that subphase which determines policy with respect to feasible global optimizations -- that is, it determines which of the feasible optimizations are also desirable. OPNDPREF is that subphase which determines the context information to be passed down. As noted above, OPNDPREF makes a preliminary determination before any subnode delaying is done, but these determinations may be modified as actual information about some subnodes becomes available. The remaining subphases all operate after all subnodes have been delayed. OPTRSEL selects the operator(s), if any, to be used. CSX and RUX determine cost estimates: CSX is a "code size complexity" measure and RUX is a "register use complexity" measure. EVO determines the evaluation order of subnodes.

In the following sections both Bliss and the PDP-11 will be used extensively; for the reader who may not be familiar with either or both, brief appendices have been included.

3.1 Determination of Desirable Feasible Optimizations

Most of the feasible optimizations detected by FLO are also desirable, but not all are. We shall illustrate using various *cse*'s. Consider, for example, the simple assignment expression

X ← .A

where ".A" is a *cse*. The default policy for handling common subexpressions is to explicitly compute their value in a temporary, usually a register. Doing so, however, many not always produce the optimal code. For example, if there are only two uses of the *cse*, e.g.,

... X←.A; ...; Y←.A

following the default practice would produce

```
MOV  A,R0
MOV  R0,X
  ...
MOV  R0,Y
```

while if we simply ignore the fact that ".A" is a *cse*, we get

```
MOV  A,X
  ...
MOV  A,Y
```

As it happens, both these sequences occupy the same amount of code space (6 words). However, the second alternative has two advantages -- it involves one fewer instruction executions, and it avoids tying up a register over some span of the program (registers are a scarce resource on the PDP-11). However, yet another possibility exists. We can treat the address of A as a *cse* rather than its value; doing so would produce

```
MOV  #A,R0
MOV  @R0,X
  ...
MOV  @R0,Y
```

This alternative has the advantage that although no other occurrences of ".A" may be common subexpressions of the instances in question they may still be addressed by "@R0" -- as indeed can any instances of "A" on the left hand side of an assignment.

The following more interesting example illustrates these issues when the *cse* in question is more complex and/or when there are more than two uses of the *cse*. Consider

X ← ..(.A+4)

where "..(.A+4)" is a *cse*. Adhering to the usual strategy of loading the value of a *cse* into a register would produce the following code:

```
MOV  A,R0
MOV  @4(R0),R0
   ...
MOV  R0,X
   ...
```

We might, however, have produced the following code:

```
MOV  A,R0
MOV  4(R0),R0
   ...
MOV  @R0,X
```

Or the following:

```
MOV  A,R0
   ...
MOV  @4(R0),X
```

Which of these alternatives is best depends upon more global context. The first alternative requires four words of code to form "..(.A+4)" in a register and no additional code space for each use. The second alternative involves the same amount of code but involves an additional memory reference for each access. The third alternative involves only two words initially but requires

an extra word of code for each reference. If there are only two references to "..(.A+4)" all three alternatives require the same amount of code, but the third is best since it requires one fewer instruction executions. If ".(.A+4)" is also a *cse*, the second alternative is best in terms of code size and requires one fewer register.

As can be seen from these examples, these policy decisions are based solely on the number of instances of a node and its subnodes. Hence the appropriate decision is independent of the other delaying actions and can be performed on the way down the tree.

3.2 Determination of Evaluation Order

Consider the following arithmetic expression: a*b+(c+d)/(e+f). A simple left-to-right code generation scheme would produce code similar to the following:

```
a*b → R1
c+d → R2
e+f → R3
R2/R3 → R2
R1+R2 → R1
```

However, the following sequence will produce the same result and requires one fewer register:

```
c+d → R1
e+f → R2
R1/R2 → R1
a*b → R2
R2+R1 → R2
```

Obviously since (c+d)/(e+f) requires two registers but produces its result in a single register, the second of these may be used for the evaluation of a*b. Several authors [Nak67, Red69, Set70] have noted the advantages of rearranging the evaluation order in such cases and have given essentially equivalent algorithms for determining an evaluation order which minimizes the maximum number of registers used. These algorithms label

each node with the number of registers necessary to evaluate that node (including its subnodes) and then specify that the preferred evaluation order (for a binary operator) is to evaluate that subnode which requires the maximum number of registers first, thus leaving enough registers free to evaluate the other subnode.

The labeling algorithm is simple. Consider a binary expression ϵ of the form "ϵ_1<binop>ϵ_2" and let $r(\epsilon_i)$ be the number of registers required to evaluate ϵ_i. Then $r(\epsilon)$, the number of registers required to evaluate the entire expression is:

1) $r(\epsilon_1) \neq r(\epsilon_2)$
   $r(\epsilon) = \max(r(\epsilon_1), r(\epsilon_2))$. In this case the operand requiring the larger number of registers should be evaluated first, leaving $r(\epsilon)-1$ registers free. Since the other operand requires at most $r(\epsilon)-1$ registers, no additional registers are required. The entire expression can be evaluated using only $r(\epsilon)$ registers.
2) $r(\epsilon_1) = r(\epsilon_2)$
   $r(\epsilon) = r(\epsilon_1)+1$. In principle either expression may be evaluated first. The first expression evaluated leaves its result in a register; thus only $r(\epsilon_1)-1$ will be left free. Since the other subnode also requires $r(\epsilon_1)$ registers the total number of registers required will be $r(\epsilon_1)+1$.

Unfortunately the algorithm described above does not account for the possibility of *cse*'s. In fact, no computationally reasonable algorithm (i.e., other than complete enumeration) is currently known to determine optimal evaluation order when *cse*'s exist. Therefore the Bliss/11 compiler uses an approximate technique which includes the scheme above as a degenerate case. Although the following analysis suggests the approximate technique, but one must remember that the analysis is built on sand.

Consider a node ϵ_0 with subnodes ϵ_1 and ϵ_2. Assume that the evaluation of ϵ_1 will require three registers but, because it involves the last use of two *cse*'s, the evaluation also results in freeing two registers. Further, assume that the evaluation of ϵ_2 requires four registers and does not involve any *cse*'s. Finally,

assume that no new *cse*'s are generated. The situation is characterized by the diagram in Figure 14. Prior to the evaluation of ϵ_0 some number of registers, r, will be occupied and following the evaluation, independent of the order of evaluation of ϵ_1 and ϵ_2, r-1 (=r-2+1) registers will be occupied. (Two registers are freed by the evaluation of ϵ_1, but one extra is needed to hold the value of ϵ_0.) Our objective is to minimize the maximum number of registers, k, needed at any point during the evaluation process.

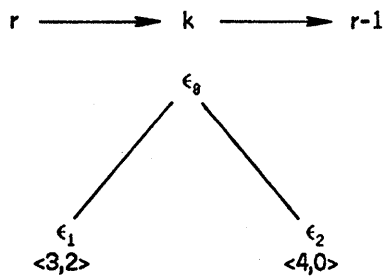


Figure 14. Determination of number of registers required to evaluate ϵ_0.

The algorithm described above, which does not account for common subexpressions, would specify that ϵ_2 should be evaluated first; this results in k = r+4. However, if ϵ_1 is evaluated first, only r-1 registers will be in use after its evaluation; hence we obtain k = r+3.

The example suggests that for each node, ϵ, if we knew how many *cse* values were "created" (first use) and "deleted" (last use) below that node, a precise determination of the optimal evaluation order would be possible. (This of course is the rub. We can't know how many *cse*'s will be created or deleted until we know the evaluation order.) Suppose each node, ϵ_i, were labeled with a triple $<u_i,c_i,d_i>$, where:

u_i = Registers "required" for the evaluation of ϵ_i in the same <u>intuitive</u> sense as above; the derivation of the form u_i is the purpose of the following analysis.

c_i = Total number of *cse*'s "created" during the evaluation of ϵ_i.

d_i = Total number of *cse*'s "deleted" during the evaluation of ϵ_i.

Now consider a binary expression with nodes labeled as shown in Figure 15. Prior to the evaluation of ϵ_0, some number of registers, r, will be "in use," i.e., holding the values of various partial results and/or *cse*'s. After the evaluation of ϵ_0, independent of the order of evaluation of ϵ_1 and ϵ_2, $r+1+(c_1+c_2-d_1-d_2)$ registers will be in use. At some point during the evaluation of ϵ_0 and its descendants, a maximum number of registers, r+k, will be in use; in general the value of k does depend upon the order of evaluation and this is the value we wish to minimize.

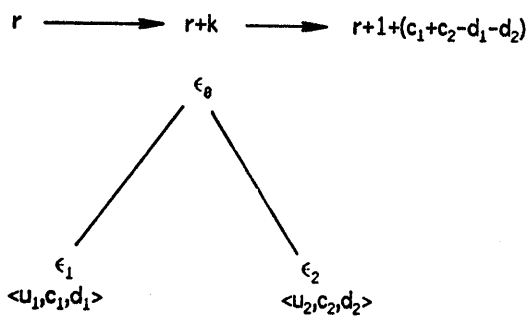


Figure 15. Determination of number of registers required to evaluate ϵ_0.

Under the assumption that c_i and d_i are known, which is equivalent to saying that the first and last uses of all common subexpressions are known, the number of registers that will be in use after each of ϵ_1 and ϵ_2 are evaluated is:

$$I = \delta(\epsilon_1) + \delta(\epsilon_2) + (c_1-d_1) + (c_2-d_2)$$

where

$$\delta(\epsilon) = \begin{cases} 0 & \text{If } \epsilon \text{ is a terminal node.} \\ 1 & \text{if } \epsilon \text{ is a non-terminal node.} \end{cases}$$

The number of registers required to evaluate ϵ_0 if ϵ_1 is evaluated first, u_{12}, and the number required if ϵ_2 is evaluated first, u_{21}, are then given by

$$u_{12} = \max(u_1, u_2+1, I+\eta(\epsilon_1,\epsilon_2))$$
$$u_{21} = \max(u_1+1, u_2, I+\eta(\epsilon_1,\epsilon_2))$$

where

$$\eta(\epsilon_1,\epsilon_2) = \begin{cases} 1 & \text{If both } \epsilon_1 \text{ and } \epsilon_2 \text{ are terminal nodes and/or } cse\text{'s which are not last uses (i.e., neither register may be reused).} \\ 0 & \text{Otherwise.} \end{cases}$$

Clearly we will choose the evaluation order depending upon whether u_{12} is smaller than u_{21} or vice versa. The labeling scheme is then

$$u_0 = \min[\max(u_1, u_2+1, I+\eta(\epsilon_1,\epsilon_2)), \max(u_1+1, u_2, I+\eta(\epsilon_1,\epsilon_2))]$$
$$c_0 = c_1+c_2+\gamma(\epsilon_1)+\gamma(\epsilon_2)$$
$$d_0 = d_1+d_2 + \lambda(\epsilon_1)+\lambda(\epsilon_2)$$

where

$$\gamma(\epsilon) = \begin{cases} 1 & \text{If } \epsilon \text{ is a } cse \text{ creation.} \\ 0 & \text{Otherwise.} \end{cases}$$
$$\lambda(\epsilon) = \begin{cases} 1 & \text{If } \epsilon \text{ is the "last use" of a } cse. \\ 0 & \text{otherwise.} \end{cases}$$

<u>Unfortunately</u> this scheme cannot be used. It depends on knowing which are the first and last uses of *cse*'s. This information cannot be known until the evaluation order is fixed, and hence cannot be used to determine the evaluation order. The analysis does, however, suggest the approximate technique used in the compiler.

For various reasons the compiler requires that the creation of a *cse* value be done by that instance of the expression which occurs earliest in the initial order; hence c_i is known. It does not,

on the other hand, specify which is the last use; hence d_i cannot be determined exactly. However, if n_i is the number of occurrences of a *cse*, then $1/(n_i-1)$ is the probability that any particular use is the last. The compiler uses exactly the algorithm implied by the analysis above except that it substitutes

$$\lambda(\epsilon_i) = 1/(n_i-1) \text{ If } \epsilon_i \text{ is a } cse \text{ \underline{use} (not creation).}$$
$$0 \text{ Otherwise.}$$

It should be noted that the algorithm degenerates to the one given earlier in the case that no *cse*'s are involved.

A different set of evaluation order decisions is based on a measure of the code complexity of the subnodes. We shall consider two cases: the order of evaluation of the operands in a boolean expression and the order of placement of the <u>then</u> and <u>else</u> portions of a conditional expression.

The semantics of Bliss specify that: (1) the operands of a boolean (notably <u>and</u> and <u>or</u>) may be evaluated in any order, and (2) a boolean which produces a flow result (e.g., in the boolean expression of a conditonal) need only be evaluated as far as necessary to determine truth or falsity of the expression. Therefore, in a construct of the form "<u>if</u> (ϵ_1 <u>and</u> ϵ_2) <u>then</u>..." it is advantageous to first evaluate whichever of ϵ_1 and ϵ_2 is most likely to determine the falsity of "ϵ_1 <u>and</u> ϵ_2" with the least effort. Lacking knowledge of the probability of the truth or falsity of the expressions ϵ_1 and ϵ_2 independently, the compiler simply chooses to evaluate the least complex one first. Note that the issue of register use complexity is irrelevant here since the <u>and</u> itself need never be formed. The complexity measure used is the (approximate) amount of code required to evaluate the operands.

The peculiarities of the target machine give rise to another use of the code complexity. Consider the conditional expression

$$\underline{\text{if}}\ \epsilon_0\ \underline{\text{then}}\ \epsilon_1\ \underline{\text{else}}\ \epsilon_2$$

which is obviously equivalent to

$$\underline{\text{if}}\ \underline{\text{not}}\ \epsilon_0\ \underline{\text{then}}\ \epsilon_2\ \underline{\text{else}}\ \epsilon_1$$

Normally one would not expect one of these forms to be

preferable to the other (unless, of course, a symmetric set of relational operators were not available). However, the fact that the conditional branch instruction on the PDP-11 can branch only a short distance makes whichever form places the smaller (less complex) code segment in the then portion more advantageous.

3.3 Target Paths and Unary Complement Operators

These two topics will be discussed together since they are frequently, but not always, related. In particular, we shall choose our examples in this section from the plus "+" operator where the two are inextricably intertwined.

Consider a binary expression, ϵ_0, whose operands are ϵ_1 and ϵ_2. Suppose that the results of ϵ_1 and ϵ_2 are available in temporary locations (registers) t_1 and t_2 respectively and that the results of the entire expression are to be produced in t_0. If the operator of ϵ_0, call it op_0, is commutative, two instruction sequences are possible:*

$$(1)\quad t_1 \rightarrow t_0;\ t_0\ op_0\ t_2 \rightarrow t_0$$

or

$$(2)\quad t_2 \rightarrow t_0;\ t_0\ op_0\ t_1 \rightarrow t_0$$

We refer to a subnode as lying on the "target path" of its ancestor if the value of that node is first moved to the ancestor's temporary and then operated upon by its sibling; thus in (1), ϵ_1 is on the target path.**

The term "unary complement operators" seems to have been introduced by [Fra70] and is derived from the property that negate and complement are their own inverses, e.g., -(-a) = a. However, the term is also applied to those cases in which one of these operators may be subsumed into a binary operator at a higher level in the tree. More formal treatments than will be

---

* In this section we are purposely avoiding the question of whether ϵ_1 or ϵ_2 is evaluated first; see Section 3.2.

** It is clearly advantageous for a node on the target path and its ancestor to use the same temporary location (in which case the move can be eliminated); see Section 4.1.

given here may be found in [Set70] and [Bea72]. Identities such as the following are used:

(1) $x-y \equiv x+(-y)$
(2) $x-y \equiv -(y-x)$ x,y are real (or integer) values
(3) $x*y \equiv (-x)*(-y)$
(4) $a \wedge b \equiv \sim(\sim a \vee \sim b)$ a,b are boolean values.

The set of identities exploited may also be extended to include those related to the machine representation of values such as, for two's complement machines, $-a = \sim a+1$.

To see the effect of these transformations, consider the expression "-y*(a-b)". A simple left-to-right evaluation of this expression would produce code of the form:

```
y → R1
-R1 → R1
a → R2
R2-b → R2
R1*R2 → R1
```

By applying identities (2) and (3) above it is possible to rewrite this expression as y*(b-a), which produces

```
y → R1
b → R2
R2-a → R2
R1*R2 → R1
```

By further changing the evaluation order to "(b-a)*y" (as described above) we obtain

```
b → R1
R1-a → R1
R1*y → R1
```

In order to apply these transformations it is not necessary to perform actual transformations of the tree; rather, a set of fields may be added to the tree to specify which, if any, transformations have been performed. In the remainder of this

section we shall concentrate on the binary plus operator and its interaction with the unary negate, and with *cse*'s. For these transformations the following set of fields are adequate:

NEG (1 bit) =	0	The result is "correct," i.e., has the same sign as specified in the source program.
	1	The result is the negative of that specified in the source program.
NDT (1 bit) =	1	The result is <u>not</u> contained in a "destroyable temporary"; this will be true of all programmer-defined variables and common subexpressions.
	0	The result is contained in a "destroyable temporary."
TPATH (1 bit) =	0	The target path is the left (natural) descendant operand.
	1	The target path is the right descendant operand.
NLOD (1 bit) =	0	Load the value of the target path operand with sign unmodified.
	1	Load the negative of the value of the target path operand.
OP		encoding of the operation to be performed

The transformations to be described will utilize the NEG and NDT fields of the operands of a node to determine the NEG, TPATH, NLOD, and OP fields of the node. To make an intelligent determination, however, it is necessary to know something about the global context in which the expression represented by the node occurs; in particular for a "+" node it is necessary to know whether, in the context of its immediate ancestor, it is preferable to develop the positive or negative of the value represented by the node. Therefore whenever a DELAY routine is called, the caller may specify its preference for various attributes of the operand. In the particular case of the sign of an operand the caller may specify: (1) the correct sign is preferred (but the

opposite sign is acceptable), (2) the opposite sign is preferred (but the correct sign is acceptable), (3) the correct sign must be generated, or (4) either is equally acceptable.

For the purposes of this section the delay routine for the plus operator has the following structure:

1) Determine sign preference for one operand (e.g., the "left" one).
2) Delay that operand, passing sign preferences from (1).
3) Determine the sign preference for the other operand.
4) Delay that operand, passing sign preference from (3).
5) Determine NEG, TPATH, NLOD, and OP based on caller's sign preference and the attributes of the operands.

A full explanation of the determination of sign preference in (1) is rather tedious but is based on considerations such as the possibility that if the caller prefers the negated result it is preferable for at least one operand, and possibly both, to be negated. Decision tables for the sign preference decisions are given in Figures 16 and 17.

After the operands of a "+" node have been delayed it is possible to use a decision table such as that shown in Figure 18 to determine the NEG, TPATH, NLOD, and OP fields of the "+" node itself. While most of this table should be obvious, a few things should be noted:

1) As noted earlier, it is advantageous for an ancestor node and its target path descendant to use the same temporary location. Although the advantage is not enforced at this point in the compiler, the prototype code shown in the code field assumes it will be; this was done to accent the situations in which an explicit move to a new temporary is required.
2) In some cases the caller's sign request cannot be satisfied without additional code. It should be clear that it never requires more code to modify the sign at a higher level in the tree; hence since these requests are not binding, the caller's preference is ignored.
3) In two cases the caller's preference could be satisfied without additional code by using a "load negative." The

GIVEN			PREF
CONTEXT	NDT_L	NDT_R	
DON'T CARE or POSITIVE PREFERRED	0	0	+
	0	1	+
	1	0	+
	1	1	+
NEGATIVE PREFERRED	0	0	-
	0	1	-
	1	0	-
	1	1	-
MUST BE POSITIVE	0	0	+
	0	1	+
	1	0	-
	1	1	+

Figure 16. Sign preference for left operand of "+".

PDP-11 does not have such an instruction, however, and hence the request is not honored. These two cases would be changed for a machine for which "load" and "load negative" incurred comparable costs.

The actual implementation of the decisions represented by Figure 18 consists of computing an index (based on the caller's sign preference, NEG_1, NDT_1, NEG_2, and NDT_2), using this index to extract the information represented in the table and storing the extracted information into the node.

GIVEN				PREF
CONTEXT	NEG_L	DT_L	DT_R	
DON'T CARE or POSITIVE PREFERRED	0	0	0	DC
	0	0	1	DC
	0	1	0	+
	0	1	1	DC
	1	0	0	+
	1	0	1	DC
	1	1	0	+
	1	1	1	-
NEGATIVE PREFERRED	0	0	0	-
	0	0	1	DC
	0	1	0	-
	0	1	1	+
	1	0	0	DC
	1	0	1	DC
	1	1	0	-
	1	1	1	DC
MUST BE POSITIVE	0	0	0	DC
	0	0	1	DC
	0	1	0	+
	0	1	1	DC
	1	0	0	+
	1	0	1	+
	1	1	0	+
	1	1	1	+

Figure 17. Sign preference for right operand of "+".

CONTEXT (REQUEST)	OPERANDS (LEFT) NEG	OPERANDS (LEFT) NDT	OPERANDS (RIGHT) NEG	OPERANDS (RIGHT) NDT	RESULT BITS NEG	RESULT BITS TPATH	RESULT BITS NLOD	RESULT BITS OP	CODE	
	0	0	0	0	0	0	0	+		$t_1+t_2\rightarrow t_1$
	0	0	0	1	0	0	0	+		$t_1+t_2\rightarrow t_1$
	0	0	1	0	0	0	0	-		$t_1-t_2\rightarrow t_1$
DON'T CARE	0	0	1	1	0	0	0	-		$t_1-t_2\rightarrow t_1$
	0	1	0	0	0	1	0	+		$t_2+t_1\rightarrow t_2$
and	0	1	0	1	0	0	0	+	$t_1\rightarrow t_0$	$t_0+t_2\rightarrow t_0$
	0	1	1	0	1	1	0	-		$t_2-t_1\rightarrow t_2$
POSITIVE	0	1	1	1	0	0	0	-	$t_1\rightarrow t_0$	$t_0-t_2\rightarrow t_0$
PREFERRED	1	0	0	0	0	1	0	-		$t_2-t_1\rightarrow t_2$
	1	0	0	1	1	0	0	-		$t_1-t_2\rightarrow t_1$
	1	0	1	0	1	0	0	+		$t_1+t_2\rightarrow t_1$
	1	0	1	1	1	0	0	+		$t_1+t_2\rightarrow t_1$
	1	1	0	0	0	1	0	-		$t_2-t_1\rightarrow t_2$
	1	1	0	1	0	1	0	-	$t_2\rightarrow t_0$	$t_0-t_1\rightarrow t_0$
	1	1	1	0	1	1	0	+		$t_1+t_2\rightarrow t_2$
	1	1	1	1	1	0	0	+	$t_1\rightarrow t_0$	$t_0+t_2\rightarrow t_0$
	0	0	0	0	0	0	0	+		$t_1+t_2\rightarrow t_1$
	0	0	0	1	0	0	0	+		$t_1+t_2\rightarrow t_1$
	0	0	1	0	1	1	0	-		$t_2-t_1\rightarrow t_2$
	0	0	1	1	0	0	0	-		$t_1-t_2\rightarrow t_1$
	0	1	0	0	0	1	0	+		$t_2+t_1\rightarrow t_2$
	0	1	0	1	0	0	0	+	$t_1\rightarrow t_0$	$t_0+t_2\rightarrow t_0$
NEGATIVE	0	1	1	0	1	1	0	-		$t_2-t_1\rightarrow t_2$
PREFERRED	0	1	1	1	1	1	0	-	$t_2\rightarrow t_0$	$t_0-t_1\rightarrow t_0$
	1	0	0	0	1	0	0	-		$t_1-t_2\rightarrow t_1$
	1	0	0	1	1	0	0	-		$t_1-t_2\rightarrow t_1$
	1	0	1	0	1	0	0	+		$t_1+t_2\rightarrow t_1$
	1	0	1	1	1	0	0	+		$t_1+t_2\rightarrow t_1$
	1	1	0	0	0	1	0	-		$t_2-t_1\rightarrow t_2$
	1	1	0	1	1	0	0	-	$t_1\rightarrow t_0$	$t_0-t_2\rightarrow t_0$
	1	1	1	0	1	1	0	+		$t_2+t_1\rightarrow t_2$
	1	1	1	1	1	0	0	+	$t_1\rightarrow t_0$	$t_0+t_2\rightarrow t_0$
	0	0	0	0	0	0	0	+		$t_1+t_2\rightarrow t_1$
	0	0	0	1	0	0	0	+		$t_1+t_2\rightarrow t_1$
	0	0	1	0	0	0	0	-		$t_1-t_2\rightarrow t_1$
	0	0	1	1	0	0	0	-		$t_1-t_2\rightarrow t_1$
	0	1	0	0	0	1	0	+		$t_2+t_1\rightarrow t_2$
MUST BE	0	1	0	1	0	0	0	+	$t_1\rightarrow t_0$	$t_0+t_2\rightarrow t_0$
POSITIVE	0	1	1	0	0	0	0	-	$t_1\rightarrow t_0$	$t_0-t_2\rightarrow t_0$
	0	1	1	1	0	0	0	-	$t_1\rightarrow t_0$	$t_0-t_2\rightarrow t_0$
	1	0	0	0	0	1	0	-		$t_2-t_1\rightarrow t_2$
	1	0	0	1	0	1	0	-	$t_2\rightarrow t_0$	$t_0-t_1\rightarrow t_0$
	1	0	1	0	0	0	1	-	$-t_1\rightarrow t_0$	$t_0-t_2\rightarrow t_0$
	1	0	1	1	0	0	1	-	$-t_1\rightarrow t_0$	$t_0-t_2\rightarrow t_0$
	1	1	0	0	0	1	0	-		$t_2-t_1\rightarrow t_1$
	1	1	0	1	0	1	0	-	$t_2\rightarrow t_0$	$t_0-t_1\rightarrow t_0$
	1	1	1	0	0	1	1	-	$-t_2\rightarrow t_0$	$t_0-t_1\rightarrow t_0$
	1	1	1	1	0	0	1	-	$-t_1\rightarrow t_0$	$t_0-t_2\rightarrow t_0$

Figure 18. Target path and unary complement decision for "+".

3.4 Other Delaying of the Plus Operator

Other delaying operations are possible on "+" nodes besides those described in the previous section; in particular, the indexing portion of the effective address calculation hardware provides a potential implicit addition. Exploitation of such operations is highly machine-specific. Thus while optimizations analogous to those described in this section are possible on most machines, the details will vary widely (on machines such as the IBM 360, for example, where double indexing is possible).

To see the effect of these optimizations, consider the following trivial program:

```
(EXTERNAL A,B,C; A←.(A+.B+3+.C)+4)
```

The code produced by the compiler for this example is:

```
MOV  C,R5
ADD  B,R5
MOV  A+3(R5),R5
ADD  #4,R5
MOV  R5,A
```

Examination of this example will show that both "A+" and "+3" were performed implicitly by indexing in the MOV instruction.

In order to perform this type of optimization two kinds of information are necessary -- the context in which the expression occurs and the "state" of similar optimizations performed on its operands. Once again, although some of the optimizations are applicable to other operators, we shall focus on the plus operator for a concrete example.

Because of asymmetries in the PDP-11 addressing structure -- and, indeed, in that of most machines -- it is desirable to distinguish two contexts in which an expression may occur; that of an <u>operand</u> and that of an <u>address</u>. Consider the expression ".A+4" in the two examples below and the difference in the code produced for each:

```
(a)  X ← .A+4                    (b)  X ← .(.A+4)

     MOV A, X                         MOV A,R0
     ADD #4, X                        MOV 4(R0), X
```

In (a) the expression ".A+4" is an operand and its value must actually be generated. In (b) the value of ".A+4" is not needed explicitly -- only implicitly in forming an address -- and thus may be formed when needed by the indexing operation.

Since the ancestor of a node is aware of the context in which its descendant occurs, context information is passed down from ancestor to descendant by the mechanism which should be familiar by now. The descendant, in turn, passes context information to its descendants and generates code (if any) as appropriate on the basis of the state of its descendants, sets its own state and returns to the ancestor. In the particular case of the address vs. operand context, four context specifications are used:

1) operand — Code must be generated, as in example (a) above, to place the value of the expression in an addressable location.
2) temporary — This is essentially equivalent to the "operand" case except that the value must be generated in a compiler-generated temporary, i.e., a register. This case is used almost exclusively for frequently used *cse*'s.
3) address — Exploitation of indexing, etc., is possible and desirable.
4) arbitrary — Either form is allowed and the called routine is free to generate the "least expensive" form; the caller will generate appropriate code if it doesn't like what it gets back.

The delaying actions in the case that an "operand" (or "temporary") is requested are controlled exclusively by the target path, unary complement operators, and evaluation order discussed previously; here we will focus on "address" (or "arbitrary") requests. For these there are six "interesting" states which correspond, roughly, to addressing modes which may be optimized further; they are:

L	literal	These are simple literal values.
RN	relocatable name	These are names which will have an absolute address when the program is loaded. They would be equivalent to literal values except they involve an unknown, and unknowable, relocation constant.
T	temporary	The value of the node is contained in a temporary, i.e., a register.
TL	temp + literal	The value of the node is the result of adding a literal to a value contained in a temporary -- but that literal has not been explicitly added yet, i.e., indexing is indicated.
TRN	temp + relocatable name	Similar to TL except, again, for the unknown relocation constant.
Z	other	All other cases.

The state for an expression is determined from the states of its operands. Rules for deriving these states are given in Figure 19. Each entry in this table contains three pieces of information (the first and/or third of which may be empty): the code produced (if any), the new state, and the compile-time operations necessary to derive the parameters of the new state. In order to describe these attributes of the state transition; T_0, T_1, and T_2 are used to name the temporary of the node and its left and right operands respectively; L_0, L_1, and L_2 denote the corresponding literal parameters associated with the T and TL states; and N_0, N_1, and N_2 denote the corresponding relocatable names associated with the RN and TRN states. For example, consider the expression ϵ_0 , which is actually "$\epsilon_1+\epsilon_2$", and assume that the context for ϵ_0 requires an "address." Further, suppose that the states of ϵ_1 and ϵ_2 are TL and TRN respectively. Then the table specifies that the contents of the two temporaries should be added together and the resulting state should be TRN with the resulting name differing by a (known) constant amount (L_1).

The effect of these transformations in a larger context may be seen by returning to the example introduced at the beginning of this section:

LEFT OPERAND STATES		RIGHT OPERAND STATES					
		L L_2	RN N_2	T $.T_2$	TL $.T_2+L_2$	TRN $.T_2+N_2$	Z Z_2
L	L_1	L $L_0=L_1+L_2$	RN $R_0=N_2+L_1$	TL $L_0=L_1$	TL $L_0=L_1+L_2$	TRN $N_0=N_2+L_1$	$T_0 \leftarrow Z_2$ TL $L_0=L_1$
RN	N_1	RN $N_0=N_1-L_2$	$T_0 \leftarrow N_1$ TRN $N_0=N_2$	TRN $N_0=N_1$	TRN $N_0=N_1+L_2$	$T_0 \leftarrow .T_2+N_2$ TRN $N_0=N_1$	$T_0 \leftarrow Z_2$ TRN $N_0=N_1$
T	$.T_1$	TL $L_0=L_2$	TRN	$T_0 \leftarrow .T_1+.T_2$ T	$T_0 \leftarrow .T_1+.T_2$ TL $L_0=L_2$	$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=N_2$	$T_0 \leftarrow .T_1+Z_2$ T
TL	$.T_1+L_1$	TL $L_0=L_1+L_2$	TRN $N_0=N_2+L_1$	$T_0 \leftarrow .T_1+.T_2$ TL $L_0=L_1$	$T_0 \leftarrow .T_1+.T_2$ TL $L_0=L_1+L_2$	$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=L_1+N_2$	$T_0 \leftarrow .T_1+Z_2$ TL $L_0=L_1$
TRN	$.T_1+N_1$	$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=N_1+L_2$	$T_0 \leftarrow .T_1+N_1$ TRN $N_0=N_2$	$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=N_1$	$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=N_1+L_2$	$T_0 \leftarrow .T_1+.T_2+N_1$ TRN $N_0=N_2$	$T_0 \leftarrow .T_1+Z_2$ TRN $N_0=N_1$
Z	Z_1	$T_0 \leftarrow Z_1$ TL $L_0=L_2$	$T_0 \leftarrow Z_1$ TRN $N_0=N_2$	$T_0 \leftarrow .T_2+Z_1$ T	$T_0 \leftarrow .T_2+Z_1$ TL $L_0=L_2$	$T_0 \leftarrow .T_2+Z_1$ TRN $N_0=N_2$	$T_0 \leftarrow Z_1+Z_2$ T

Figure 19. Exploitation of addressing modes in a "+" node.

(EXTERNAL A,B,C; A ← .(A + .B + 3 + .C) + 4)

The tree of the assignment expression is shown in Figure 20 with its arcs labeled by the context information which would be passed down (a ≡ address, o ≡ operand, b ≡ arbitrary). The states of the nodes and code generated for each are shown in Figure 21. It should be noted that:

1) New temporaries (T_1,T_2,T_3) are introduced only where explicit code is generated. Otherwise the temporaries are assumed to propagate up the tree. (In fact, comparison of the tree with the code shown earlier shows that all the temporaries have been bound to the same register, R5.)

2) Note that the transition of the highest "+" node is different -- the transition is not given in our table because the "+" node receives an "operand" request from the "←" node.

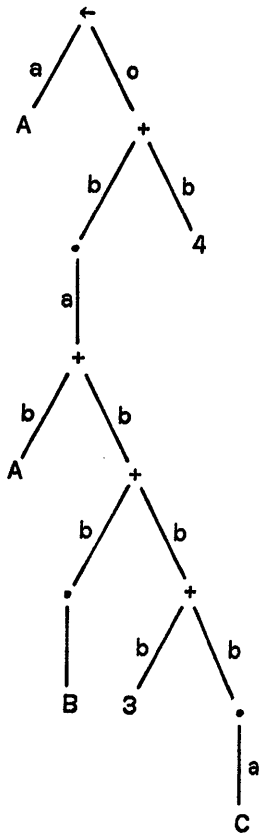


Figure 20. Tree of example program with context information.

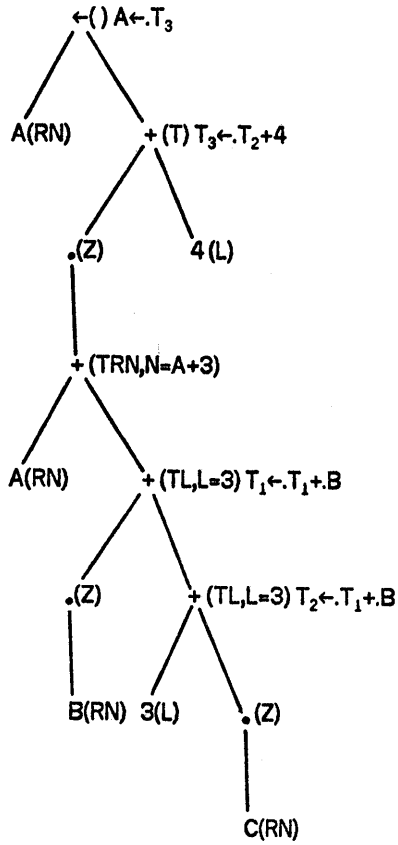


Figure 21. Tree of example program with state information.

3.5 Other Functions of DELAY

Much of the earlier material on DELAY has concentrated on the binary "plus" operator. This is due both to the importance of the particular operator and because it suggests how similar optimizations are done for other operators. Thus other operators concern themselves with determining desirable optimizations, evaluation order, unary complement operations, etc. However, there are a number of other DELAY optimizations for other

operators which are not covered by the earlier discussions; we list some of them below:

1) Converting multiply to shift. On most machines it is more economical to perform some integer multiplies by equivalent sequences of shift and add instructions. This is especially true on the PDP-11 and other minicomputers which do not have a multiply instruction.

2) Distributed multiply. In some, but not all, cases it is desirable to distribute multiplication across addition. For example, the typical Fortran-style two dimensional array access is of the form

```
A + (.I-1)*n + (.J-1)
```

where n is a constant at compile-time. In the case n=8, distributing the multiply and applying the other DELAY optimizations converts this expression into

```
A-9 + .I↑3 + .J
```

3) Constant values. Most constant folding is done during syntax analysis and tree generation; however, there are expressions whose value is constant even though the expression itself -- or at least some of its operands -- must be evaluated. For example, the expression

```
(X←.Y) and 0
```

clearly has a value of zero. However, since it has side effects it cannot be eliminated. DELAY exploits these constant cases.

Chapter 4

TNBIND

The compiler creates a "temporary name" or TN to refer to any temporary storage location. TN's are assigned not only to intermediate results generated during expression evaluation, but also to instances of *cse*'s and to (some) user declared variables. The classic register allocation problem is thus cast as binding each TN to some available register or memory location.

It is important to note that the instruction set of the PDP-11 does not require that registers be used. Any machine instruction will accept both source and destination operands in memory; the registers, aside from being faster, add only the convenience of additional addressing modes (e.g., indexing). They do not allow any operation which could not be performed without registers at some additional cost. Thus the problem of having insufficient registers cannot occur; rather, the problem is that of finding a set of bindings for a large number of temporary names to a small number of available registers and memory locations so that program size and execution time are minimized. This binding is performed by the phases of the compiler in the TNBIND module.

4.1 TLA

The discussion of TLA is divided into five pieces: targeting, cost determination, lifetime characterization, the runtime stack, and label assignment.

4.1.1 Targeting

TLA is organized as a set of mutually-recursive, node-specific action routines which perform an execution-order tree walk. The appropriate action routine is invoked at each node and is passed a (possibly null) TN as its "target" from its ancestor. If possible and reasonable, the node-specific routine will use this target TN for the result of its node; it does so by passing the target TN as a target to the routine which handles its target path subnode. Upon return from the action-specific routines for its subnodes, which will assign TN's for their results, it is necessary to resolve the target request from above with what was actually done below. In the simple case, the target TN will have been assigned to the target path subnode and that TN will be used for the current node as well. Other cases are more complex and will be discussed below. Before considering the details, however, a simple example is in order. Consider the following program:

(<u>own</u> x, y; <u>routine</u> f(z) = <u>begin</u> local l; l ← .x + .z; .l <u>end</u>; ...)

The tree for the routine "f" is shown in Figure 22. For convenience the nodes in this tree have been labeled in the order in which they would be encountered while backing out of an execution-order tree walk. In principle, of course, a unique temporary name could be associated with each node plus one additional TN for the local variable, l. To do so, however, would lose much of the structural information available in the tree. One of the prime goals of TLA is to avoid such losses.

The root node "8:routine" has no ancestor and hence is not passed a target. The node-specific action, however, knows this and also knows that the value of a routine is returned in a standard place. It therefore creates a TN, say t_0, and passes this as a target to its subnode. The node-specific action for compounds passes its target to only the last subnode. The node-specific action for store, "←", is quite complex, but in cases such as that shown will pass the location of the left hand side as a target to the right hand side. Thus after the compound, the state is characterized by the diagram in Figure 23 in which targets are shown by directed arrows adjacent to the tree and assignments are indicated in parentheses next to the node. Notice that the "compound" had to resolve the request that its result be placed

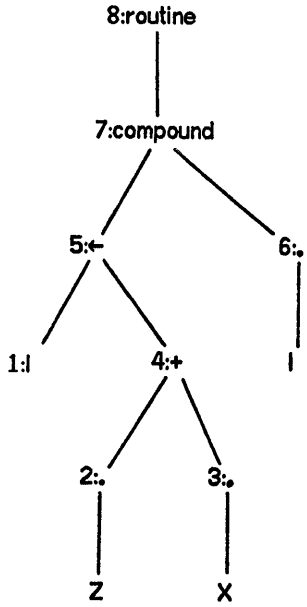


Figure 22. Tree of example routine.

---

in t_0 with the fact that its last subnode placed its result in t_1. Such resolutions are invariably made in favor of leaving the value where it is since at most one move instruction is implied and it might as well be generated higher in the tree. However, we can note at this stage that it would be "convenient" if, even though t_0 and t_1 are distinct temporary names, they were bound to the same location. Therefore another mechanism, "preferencing," is invoked; specifically the node-specific action routines for "←" will insure that the two temporaries are in the same "preference class." Later, the PACK phase will attempt to bind two or more TN's which belong to the same preference class to the same location. For this example in particular it is possible to use the same register (R0) for both the value of the routine and the local "I", and the code produced for the entire routine is:

```
MOV  Z,R0
ADD  X,R0
RTS  PC
```

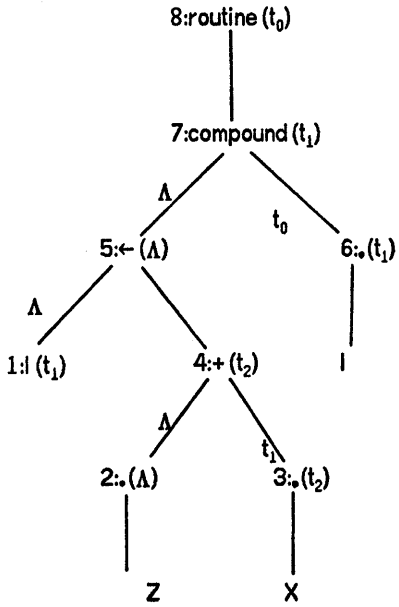


Figure 23. Tree with temporary names assigned.

With this introduction to the function of TLA we can return to the complexities of TLA alluded to earlier; these complexities arise because of node-specific idiosyncracies and common subexpressions. The generic form of node-specific actions is (roughly):

```
if ~u(node) then
    begin
    if c(node) then target ← Λ;
    {perform node-specific action,
    place actual assignment in tn};
    if c(node) then binduses (node,tn);
    end
```

where

u(node) =	True iff the node is a common subexpression <u>use</u>.
c(node) =	True iff the node is a common subexpression <u>creation</u>.
binduses	Binds all uses of a given common subexpression creation to the same TN as the creation.

Nodes which represent neither uses nor creations of common subexpressions are handled as described previously. Uses of *cse*'s, on the other hand, are simply ignored since their temporary names are assigned when the creation node is handled. Creation nodes ignore their target in order to insure that later ranking and packing treat the *cse* as an independent entity.

It is impossible to treat all of the node-specific actions here; however, two will be discussed as examples -- the "typical" binary operator and <u>if</u>-<u>then</u>-<u>else</u>. These were chosen to illustrate the difference between arithmetic and control operators. In order to present these examples some functions and predicates are needed:

<u>functions</u>

tla(node,target)	Invoke the appropriate node-specific action on the specified node, passing it the specified target TN. TLA returns a (possibly different) TN.
tllist(node)	Invoke tla on each element of the list (α, ω, χ, or ρ) named by node.
gettn	Create a new temporary name and return it.
wantpref(t_1,t_2)	Form the union of the preference classes of the temporary names t_1 and t_2.

<u>predicates</u>

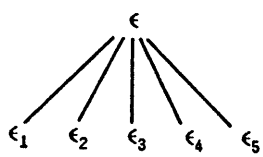
δ(node)	True iff the result specified by the node is "destroyable," i.e., is not a user-defined variable or *cse* with remaining uses.
τ(node$_i$)	True iff the ith subnode of the specified node is its "target path."
tnneeded(node)	True iff a result temporary is needed for the specified node.

Now consider a binary node, ϵ, with subnodes ϵ_1 and ϵ_2. The node-specific action is:

```
tn←target;
l←tla (ε1, if τ(ε1) then tn else Λ);
r←tla (ε2, if τ(ε2) then tn else Λ);
temp←Λ;
if tn=Λ then
   if τ(ε1) then (if δ(ε1) then tn←l else temp←l) else
   if τ(ε2) then (if δ(ε2) then tn←r else temp←r);
if tnneeded(ε) then
   (if tn=Λ then tn←gettn( ); wantpref(tn,temp));
```

Thus the action is first to perform "tla" on the subnodes, passing the target down along the target path. If there was no target, the resultant TN for the target path is chosen if it is destroyable. If on the other hand there was no target and the TN on the target path is not destroyable, a new temporary name is created and added to the preference class of the target path result.

Now consider an <u>if</u>-<u>then</u>-<u>else</u> node, ϵ. Such a node actually has five subnodes



where ϵ_2 is the boolean, ϵ_3 and ϵ_4 are the <u>then</u> and <u>else</u> parts, and ϵ_1 and ϵ_5 are the α- and ω- sets respectively. The node-specific action is:

```
tn←target;
tllist(ε1);
tla(ε2, Λ);
if tnneeded (ε) then
   begin
   if tn = Λ then tn←gettn( );
   tt←tla (ε3, tn);   te←tla (ε4, tn);
   if tn ≠ tt then wantpref (tn, tt);
   if tn ≠ te then wantpref (tn, te);
   end;
tllist(ε5);
```

The effect of this is: (1) assign TN's for the α set and the boolean; (2) if no TN is needed for the result of the expression, assign TN's for the then and else parts passing no target; (3) if a TN is needed for the expression, pass the target to the then and else parts and use wantpref if required; (4) assign TN's for the ω set.

Note that the boolean expression, ϵ_2, is passed a null target since we do not care where its result is generated.

4.1.2 Cost Determination

Each temporary name may be used more than once and may be used in each instance in one of several ways -- e.g., as a simple value, as an index quantity, etc. In the case that there are insufficient registers to hold all the temporary values needed at one time we would like those in registers to be the "most important" ones and those in memory to be of lesser importance. We have chosen to define the importance in terms of a cost measure; the measure currently used is:

$$C_{max} = \sum_{\text{accesses}} (\sigma_{max}(\text{access}) * (\text{loopdepth} + 1))$$

$$C_{min} = \sum_{\text{accesses}} (\sigma_{min}(\text{access}) * (\text{loopdepth} + 1))$$

where σ(access) is a measure of the cost of accessing the TN in terms of words of code generated and number of run-time memory references. The σ_{min} measure assumes the TN will be

bound to a register; σ_{max} assumes the TN will be bound to a memory location. For example, in the PDP-11 the indexing addressing mode requires an additional word of storage to hold the base value as well as an additional memory reference to access that value. Each level of indirection also adds a memory reference. The code for each node has a characteristic use pattern for its own temporary as well as for those of its subnodes. This use pattern may involve more than one access of any of the temporaries. The σ measures take this use pattern into account.

Notice that the term "loopdepth+1" tends to heavily weight the importance in favor of temporary names which occur inside loops. This is done simply to insure that the temporaries in the most deeply nested loops, where most execution time is spent, are more likely to be bound to registers.

4.1.3 Lifetime Determination

The characterization of the lifetime of a temporary is one of those extremely important problems for which no truly satisfactory solution exists in the literature. The simplest scheme is to characterize the lifetime in terms of a linear span from its earliest to latest use in the program. This scheme is unsatisfactory in the context of forked constructs; consider, for example,

$$\ldots \epsilon \ldots \text{ if } \beta \text{ then } \gamma \text{ else } (\ldots \epsilon \ldots)$$

The linear measure would prevent the use of the location of the expression "ϵ" throughout the entire body of "γ" even though this might be legitimate. The other extreme is to characterize the lifetime of a location along each possible flow path. While this affords a complete solution, it entails recording a great deal of data. The scheme used in the Bliss/11 compiler is a compromise which has worked satisfactorily (but we would like a better one).

During TLA each node is labeled with two values, a "linear order number" or "lon" and a "flow order number" or "fon." The lon is a unique, monotonically increasing value assigned in execution order. Thus the lon specifies a relative position in the final program where the code for a specific node will go. The fon, on the other hand, monotonically increases along a flow path

but is the same for the head of each branch emanating from a fork. The <lon,fon> pairs for a hypothetical case expression are illustrated in Figure 24.

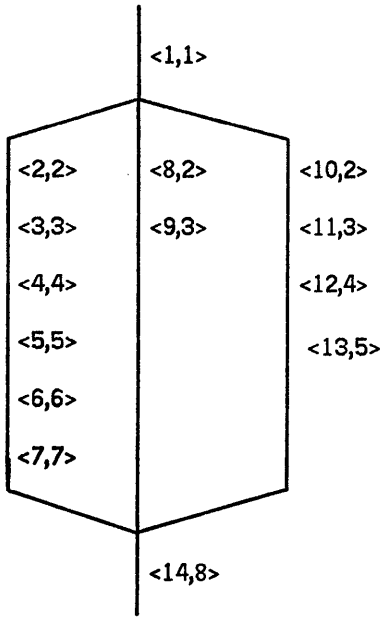


Figure 24. Lon-fon values for a case expression.

Since fon ≤ lon, the set of possible <lon,fon> pairs is representable by the set of points on or below the diagonal in the first quadrant of 2-space as shown in Figure 25a. We choose to represent the lifetime of a temporary name by four points which define a rectangle in this space -- its lon and fon values of first and last use (lonfu, lonlu, fonfu, fonlu). This lifetime may be seen graphically in Figure 25b. Two temporary names are considered contemporaneous if the rectangles representing their lifetimes intersect. In such a case they cannot be allocated to the same location.

The lonfu, lonlu, fonfu, and fonlu are noted during TLA as the temporary names are assigned to nodes by each node-specific

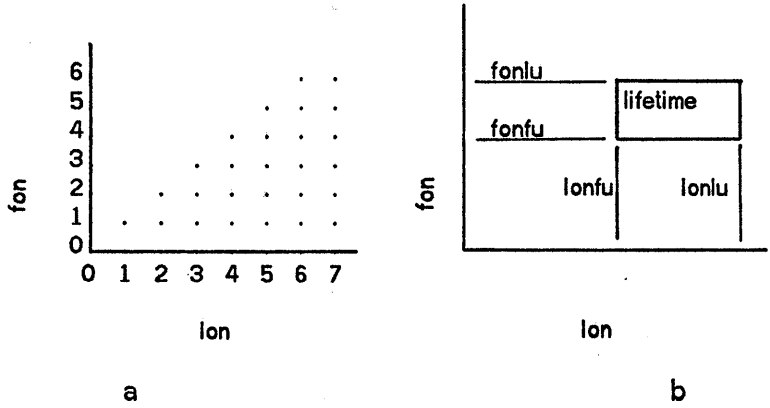


Figure 25. Graphical representation of the lon-fon space.

action routine. In particular, the subnodes of a given node are "spanned" in order to establish that the result represented by the subnode is still "in-use" at the parent.

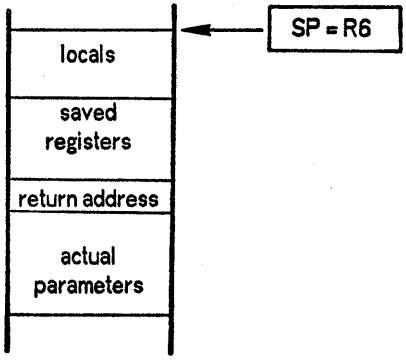


Figure 26. The runtime stack.

4.1.4 The Runtime Stack

Generally we have avoided the description of the runtime environment for compiled programs. This section will deviate from that practice in order to describe one of the more important issues treated by TNBIND. The environment of a routine consists of a stack containing (possibly) some local variables, saved registers, return address, and actual parameters of the routine as illustrated in Figure 26. A single register, SP, points to the current top of this stack; no other pointers into the stack are maintained, hence all items in the stack are indexed relative to this register. A typical routine call "f(.x,.y)" for example, produces

```
MOV  X,-(SP)
MOV  Y,-(SP)
JSR  PC,F
```

This places the actual parameters and return address on the stack and transfers control to the routine "f". The body of "f", in turn, saves registers, creates and deletes locals, etc., and pops the return address off of the stack when it returns. Now, the point of all this is that neither the called routine nor the caller removes the actual parameters -- not right away at least. Rather, the caller leaves these two locations on the stack for possible use for either temporaries or locals. The effect of this strategy is illustrated by the following example:

```
routine r(x)=
        begin local a, b;
        a←f(.x);  b←g( );
        return .a + .b
        end;
```

The code produced for this routine is:

```
MOV  X,-(SP)
JSR  PC,F
MOV  R0,@SP
JSR  PC,G
ADD  (SP)+,R0
RTS  PC
```

In this particular case the local "a" is allocated to the location "created" by the parameter push from "f(.x)." The local "b" is allocated (because of the targeting/preferencing mechanism described earlier) to R0. The PDP-11 allows the value "popped" off the stack to be an operand of any instruction thus deallocating the local at no cost in code size.

In order to exploit this optimization the compiler must keep track of the number of "dynamic temporaries" created by pushing parameters and must occasionally generate code to remove some or all of them. In particular, dynamic temporaries must be removed at points where two flow paths merge: specifically, (1) at the "bottom" of a loop, so that the stack depth will be the same for the next iteration, (2) at the "join" of a forked construct, so that the stack depth will be the same through all paths leading through the join.* Dynamic temporaries are also removed whenever the number of them seems "unreasonably large."

The "dynamic temps" are implemented by simulating the stack at compile-time. The node-specific routine for actual parameters "opens" dynamic temps for parameters. These are "closed" again at the points mentioned above, e.g., at the bottom of loops. "Open" and "closed" periods for the dynamic temporaries are represented in a manner to be described below in the section on PACK. Information is passed to the code phase to effect runtime stack adjustment in a field in each node. TLA sets this field, the "dynamic-temp-depth" field, to the depth of the simulated stack at that point in the program. The code phase of the compiler emits instructions to adjust the stack whenever its simulated stack depth differs from that set in the node.

* The number is reduced to the minimum of the number generated along any single path on those paths which generate more than this minimum number.

4.1.5 Label Assignment

Label assignment has nothing whatsoever to do with the main topic of this section. However, label assignment is performed during TLA and therefore will be discussed here -- albeit briefly. Label assignment is another targeting process. Consider, for example, an <u>if</u>-<u>then</u>-<u>else</u> expression of the form

$$\underline{\text{if}}\ (\sim\epsilon_1)\ \underline{\text{and}}\ \epsilon_2\ \underline{\text{then}}\ \epsilon_3\ \underline{\text{else}}\ \epsilon_4$$

The node-specific routine for <u>if</u> passes to the <u>and</u> node two labels -- a "true" label which is that of the <u>then</u> part, ϵ_3, and a false label which is that of the <u>else</u> part, ϵ_4. The <u>and</u> node generates no code; but the node-specific routine for <u>and</u> passes the false label to both of its operands: the true label passed to its left operand is that of its right operand, while the true label passed to the right operand is the true label input to the <u>and</u> node. The "<u>not</u>" node-specific routine also does not generate code, it merely inverts the input labels. The effect of label targeting may be seen in the following example:

if .x≤y and (.x≥1 <u>or</u> .y ≠ 0) then x←1 else y←1

which produces the following code:

```
     CMP  X,Y
     BGT  L2
     CMP  X,#1
     BGE  L1
     TST  Y
     BEQ  L2
L1:  MOV  #1,X
     BR   L3
L2:  MOV  #1,Y
L3:
```

4.2 RANK

The function of the RANK phase is simply to order all temporary names so that the PACK phase may process them in order of their importance. At the same time the temporary names are separated into four categories depending on their individual requirements for binding: (1) specific register,* (2) some register,* (3) a memory (stack) location, and (4) either a register or a memory location. The majority of TN's fall into the fourth category.

Ranking is based on both the cost measure and the lifetime characterizations mentioned earlier. Specifically the ranking is based on

$$\frac{C_{max}}{MAXC} * \frac{MAXS - (\log_2 (fonlu - fonfu + 1) - 1)}{MAXS}$$

where MAXC is the highest value of C_{max} for any TN, and MAXS is the highest value of $\log_2(fonlu-fonfu+1)-1$. The effect of this ranking criterion is to treat TN's whose uses are distributed over a large span as less important than those with similar cost measures but more compact uses.

TN's which fall into the third category are handled somewhat differently in RANK. Since they must be bound to memory locations either because of a specific user request or because their addresses are used in some non-standard context.** The max and min cost measures are meaningless; instead these TN's are ranked according to their lon of first use. This increases the probability that the first assignment to the TN may be made by a "push" operation, which is less expensive than a normal store into a stack location.

---

* Bliss programs may declare that certain variables must be allocated to a register or to a specific register.

** The PDP-11 does not allow a register to be addressed as a normal memory location. Hence a local variable whose address is passed as a parameter to another routine, for example, must be assigned a real memory location.

4.3 PACK

Before proceeding to a description of the pack algorithm itself we must discuss the mechanism by which the availability of various locations, whether registers or in memory, is represented. Specifically, each location, l_i, is represented by a set, L_i; the elements of L_i are temporary names. Initially all L_i are empty. The following sets, functions, and predicates are defined to simplify the explanation:

<u>sets</u>

The set $\{L_i\}$ is partitioned into equivalence classes according to the type of location represented:

$R = \{R_i\}$	The set of registers.
$D = \{D_i\}$	The set of "dynamic temporaries" created, as described above, by pushing actual parameters.
$S = \{S_i\}$	The set of "static temporaries"; these temporaries are allocated (on the stack) at routine entry.

<u>predicates</u>

empty(L)	True iff the set L is empty.
intersect(t_1,t_2)	True iff the lifetimes of the temporary names t_1 and t_2 intersect.
fits(t,L)	True iff $\forall\ t_1 \in L$ (~intersect(t,t_1)).
bound(t)	True iff the temporary name t has been bound to some location.

<u>functions</u>

open(L) =	<u>if</u> empty(L) <u>then</u> L←{z,i} <u>else</u> ERROR where z and i are special "dummy" temporary names that have lifetimes of <0,0,0,0> and <∞,∞,∞,∞> respectively.
close(L,lon,fon) =	L←L ∪ {c} where c is another "dummy" temporary name specially created with a lifetime of <lon,∞,fon,∞>.
reopen(L,lon,fon) =	<u>if</u> empty(L) <u>then</u> open(L) <u>else</u> <u>if</u> ∃ t ∈ L ∋ lifetime(t) = <x,∞,y,∞> <u>then</u> lifetime(t) ← <x,lon,y,fon>

tryfit(t,L) =	<u>if</u> fits (t,L) <u>then</u> (L←L ∪ {t}; <u>true</u>) <u>else</u> <u>false</u>
location(t) =	The set L of which t is an element.
P(t) =	The preference class of t.

Note: in terms of these predicates, the "stack simulation" alluded to in Section 4.1.4 is actually a stack of these sets. A set is reopened each time a simulated parameter push is done, and one or more sets are closed each time the simulated stack is cut back.

The basic packing algorithm consists of considering each temporary name and attempting to fit it into various sets. The four categories of TN's created by RANK (discussed above) are considered in the following order:

1st) Those TN's that must be bound to a specific register.
2nd) Those TN's that must be bound to some register.
3rd) The TN's which may be bound to either a register or a memory location.
4th) The TN's which must be bound to a memory (stack) location.

The algorithms for the four categories of TN's are presented as routines in Figure 27.

In the process of attempting to fit a TN into one of the open registers (or into one of the registers to which it has been "preferenced"), it may happen that the TN will not fit. In such a case it is possible that some permutation of the assignments already made would allow the new assignment to be made. The set of feasible permutations is described recursively below.

If t is a temporary name, then let t* be a "marked" version of the same TN. A "marked" TN is considered unmovable; that is, it may not be removed from the set representing the register to which it has been bound. (For example, the TN's in the specific register category discussed above are all marked.) Let

$$K(R_i,t) = Z \text{ if } \exists\, t_j \in R_i \text{ such that intersect}(t,t_j) \wedge t_j \text{ is marked,}$$
$$= \{t_j \mid t_j \in R_i \wedge \text{intersect}(t,t_j)\} \text{ otherwise.}$$

That is, $K(R_i,t)$ is the set of TN's already bound to register i which

```
routine packspecificreg(tn,n) =
    if not tryfit(tn,R_n) then ERROR;


routine packanyreg(tn) =
    begin
    ∀ r ∈ R do
        if tryfit(tn,r) then return;
    ERROR
    end;


routine packanywhere(tn)=
    begin
    ∀ t ∈ P(t) do
        if bound(t) then
            if tryfit(tn,location(t)) then return;
    ∀ r ∈ R do
        if not empty(r) then
            if tryfit(tn,r) then return;
    if trypermute(tn) then return
    ∀ r ∈ R do
        if empty(r) then
            begin open(r); tryfit(tn,r); return end;
    ∀ d ∈ D do
        if tryfit(tn,d) then return;
    {add tn to the list of TN's which must be bound to memory
    in the sequence described in RANK}
    end;


routine packmemory(tn) =
    begin
    ∀ s ∈ S do
        if tryfit(tn,s) then return;
    {open a new "static temp" s'};
    tryfit(tn,s')
    end;
```

Figure 27. Basic algorithms for PACK.

"conflict" (intersect) with t unless one of these TN's is marked in which case $K(R_i,t) = Z$. Using this function, the algorithm to attempt the other permutations is shown in Figure 28.*

```
routine trypermute(t) =
    begin
    ∀ r ∈ R do
        if K(r,t) ≠ Z then
            begin
            X ← K(r,t);   r ← (r - X) ∪ {t*};
            ∀ x ∈ X do
                begin
                X ← X - x;
                if ~trypermute(x) then (r ← (r - {t*}) ∪ X ∪ {x}; return false);
                end
            r ← (r - {t*}) ∪ {t};
            end;
    return true;
    end;
```

Figure 28. Algorithm for attempting permutations of bindings.

* In order to minimize the number of permutations attempted, the actual algorithm is slightly more complex than the one shown in the figure.

Chapter 5

CODE

It might appear that after the previous phases have been completed, code generation would be trivial; after all, redundant computations have been detected and eliminated, (near) optimal execution order has been determined, temporary variables have been carefully allocated to the available registers, etc. It would appear that the only remaining task is to perform an execution order tree walk and generate the code appropriate to each node (if, indeed, any is required). This is in fact what must be done, but would that it were as easy as this glib statement suggests!

There are few descriptions of code generators in the literature [Low69, Hop69, Gri71] - and for a very good reason. It falls to the code generator to cope with, and hopefully exploit, the idiosyncracies of the target machine. While not conceptually difficult, an honest attempt to exploit the target machine involves a great deal of special case analysis. Performing that analysis is both tedious to do and to explain, yet it is extremely important. In the final analysis the quality of the local code has a greater impact on both the size and speed of the final program than any other optimization.

Lest we bore the reader (and ourselves) too much, we shall present only three aspects of the code generator, namely generating the code to move a value from one arbitrary field to another arbitrary field, generating the code for the control of an <u>incr</u> loop (the Bliss counterpart of the Algol <u>for</u> or Fortran DO), and generating the code for boolean operations. None of the examples will be presented in full detail, nor do we claim that the examples are representative of the problems faced by other compilers or even the remainder of this one. The purpose in presenting these examples is merely to illustrate the level of detailed analysis necessary to generate good quality local code.

The general strategy of code generation is to perform an execution order tree walk invoking node-type specific routines. The mechanism used to implement this treewalk is identical to that used in DELAY and TNBIND. The responsibility of each specific routine is to generate the optimal local code for that node, given that the code for its subnodes has already been generated. That is, code is generated while "backing-out" of the tree walk.

In the exposition we shall use the following functions, predicates, and conventions:

(1) notation

$a\uparrow b$	The quantity a shifted b bit positions; b0 implies a left shift, b<0 implies a right shift.

(2) functions

$\gamma(op,a_1,a_2)$	Emit a PDP-11 instruction "op" with addresses a_1 and a_2; a_2 is omitted in the case that "op" is a unary.
α(node)	Generate an address of the form used by γ which is the location of the result represented by the node "node."
gl	Get a "compiler generated" label which is distinct from all previously generated labels.
pl(lab)	Place a label, "lab," at the current point in the instruction stream.
p(node)	Retrieve the "position," p, and
s(node)	"Size," s, of the result represented by "node" within the word whose address is given by "α(node)."
ν(opnd)	In the case that "opnd" describes a literal value, get that value; undefined otherwise.
τ(node)	Extracts the temporary location (if any) in which the result of "node" will reside at execution time; the composite function $\alpha(\tau(\text{node}))$, denoted $\alpha\tau$(node), generates the appropriate address of this temporary.
C(e)	Causes the code associated with expression e, if any, to be generated.

(3) predicates

δ(node)	True iff the result represented by "node" is contained in a "destroyable" temporary, i.e., is <u>not</u> either a user-defined variable or a common subexpression with remaining uses.
λ(opnd)	True iff the operand is a literal.
z(opnd)	True iff the operand is the literal zero, i.e., $\lambda(\text{opnd}) \wedge (\nu(\text{opnd}) = 0)$.
o(opnd,s)	True iff the operand is a literal all of whose last s bits are one, $\lambda(\text{opnd}) \wedge [((\nu(\text{opnd}) \wedge (2\uparrow s-1)) = 2\uparrow s-1]$.

(4) conventions

(a) It will be convenient to adopt a more compact node/subnode naming convention than is used in the compiler itself. In particular we shall refer to the node itself as e_0 and its subnodes as e_1, e_2, etc. (left-to-right) respectively. We shall, further, extend this subscript convention to the functions and predicates described above. Thus, for example, $\alpha_1 \equiv \alpha(e_1)$, $\nu_2 = \nu(e_2)$, $p_1 = p(e_1)$, etc.

(b) Brackets, { }, are used to enclose instructions to be generated,* thus:

$$\{\text{ADD x,y; ASH y}\} \equiv \gamma(\text{ADD},\alpha(x),\alpha(y));\ \gamma(\text{ASH},\alpha(y))$$

5.1 Generating Data Moves

The problem we wish to consider is that of generating code to move the contents of some arbitrary field in one word to an arbitrary field of another word; in terms of Bliss source code, such moves arise in contexts such as

$$x<5,3> \leftarrow .y<9,2>$$

In terms of the tree shown in Figure 29, we are concerned with generating code for e_0. No code will have been generated for either e_1 or e_2, but DELAY will set fields in e_1 and e_2 such that

* Details of the PDP-11 instruction are given in Appendix A.

α_1 is the address of x, α_2 is the address of y
$p_1 = 5$, $s_1 = 3$, $p_2 = 9$, $s_2 = 2$
etc.

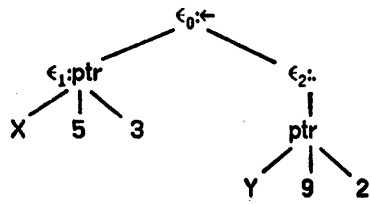


Figure 29. Tree for x<5,3> ← .y<9,2>

Since the PDP-11 does not contain instructions for general subfield moves, or even for field extraction and insertion, the effect must be achieved by an appropriate sequence of shifting, masking, etc. The variety of sequences possible for doing this is expanded significantly by the presence of the byte manipulation instructions and by the lack of instructions for multiple position shifts. Therefore, before proceeding to the description of the move generation itself we shall introduce several functions which handle the special cases of field isolation and positioning.

(1) clearfield (node, posn, size)
This function clears (sets to zero) the subfield of the operand described by "node." The following cases are treated:

(a) full word (posn = 0, size = 16). Use the CLR instruction.
(b) byte (posn = 0 or 8, size = 8). Use the CLRB instruction.*

* Here, and elsewhere in this exposition, we will ignore a level of complexity introduced by the fact that byte operations do not operate in quite the same way when the destination is a register as they do when the destination is memory.

(c) Otherwise, use the BIC instruction with an appropriate literal mask as the source operand.*

The rationale for these cases arises from the observation that CLR and CLRB are unary operators while BIC is a binary. The latter will occupy one additional word of code space and cause one additional memory reference at execution time.

(2) isolate (node, posn, size)
This function is complementary to "clearfield" in that it clears (sets to zero) everything <u>but</u> the specified subfield; it considers the same cases.

(3) dial (node, fp, fs, tp, ts)
This function moves a subfield of the result represented by "node" to another subfield in the same word.** The source subfield is described by a position and size, fp and fs respectively, and the destination subfield is similarly described by tp and ts. No special cases are treated by "dial" per se, but rather by another function "shift" described below. The action of "dial" is simply:

```
shift(node,fp,tp,min(fs,ts)); isolate(node,tp,min(fs,ts))
```

(4) shift (node, fp, tp, size)
This function shifts a subfield of a specified size from one position, fp, to another, tp. The lack of multiple bit shift or rotate instructions together with the presence of the byte operations, especially SWAB, make this a fairly complex decision; it must be based on the distance of the shift, the size of the field, and the relative and absolute positions of the source and destination subfields. The following special cases are among those recognized:

---

* Here we choose to ignore another level of complexity arising from the restriction that word instructions must specify an even word address.

** Here we are ignoring the complexity introduced by the possibility that the result represented by "node" is not "destroyable" -- i.e., it is a common subexpression whose value will be needed again (or it is a user-defined variable).

(a) $-5 \leq tp-fp \leq 4$. The straightforward option of generating a series of shifts in the intended direction is the best that can be done.

(b) $8 \leq abs(tp-fp) \leq 12$. Best here is to generate a SWAB followed by a series of shifts in the intended direction.

(c) $abs(tp-fp) > 12$. The rotate instructions are used to rotate the field in the opposite direction from that specified, hence out one end of the word and in the other. The rotate operations move the carry bit around as if it were part of the word or byte being rotated, hence one more instruction than might otherwise be expected must be generated.

(d) $5 \leq abs(tp-fp) \leq 7$. We wish to generate a SWAB and a series of shifts in the direction opposite to the one intended. Complications are introduced here by the possibility that the source or destination field overlaps both bytes of the word, in which case we should either wait to generate the SWAB until enough shifts have been generated to move the field wholly within one byte, or if the field is too large for one byte, we should use rotates rather than shifts and worry a little harder than in part (c) about the carry bit. The algorithm is:

(1) If size > 8, generate a SWAB, then a byte rotate (RORB or ROLB) to accommodate the carry bit by putting it in a harmless place, and finally a series of full word rotates.

(2) If size $\leq$ 8, generate a series of shifts (in the prescribed direction), inserting a SWAB as soon as the field is wholly within one byte (which condition must occur sometime, or the intended shift would move the field past a word boundary).

A few examples of the code generated for various shifts will illustrate some of these special cases; suppose α is the address of a word to be shifted and the original, or "from" position, the destination or "to" position, and size of the field to be shifted are

given by "fp", "tp", and "size". The code for various cases is given below:

fp	tp	Size	code
2	5	9	{ASL α; ASL α; ASL α}
12	1	3	{SWAB α; ASR α; ASR α; ASR α}
1	14	2	{ROR α; ROR α; ROR α; ROR α}
1	7	9	{SWAB α; RORB α; ROR α; ROR α}
7	1	9	{ROL α; ROL α; ROLB α; SWAB α}
1	7	8	{ASR α; SWAB α; ASR α}
2	8	8	{ASR α; ASR α; SWAB α}

We shall introduce one more function before proceeding to a description of the move operation. This function tests a single bit in a word and generates a branch to a specified label if that bit is set to zero.

(5) bittest (node, bit, lab)

This function generates a test of the specified bit in the result specified by a node and will generate a branch to the specified label which will be taken if the bit is zero at execution time. The following cases are treated:

(a)	bit = 15:	{TST α_0; BPL lab}
(b)	bit = 7:	{TSTB α_0; BPL lab}
(c)	bit = 0 $\wedge$ δ(node):	{ROR α_0; BHIS lab}
(d)	bit = 14 $\wedge$ δ(node):	{ASL α_0; BPL lab}
(e)	bit = 6 $\wedge$ δ(node):	{ASLB α_0; BPL lab}
(f)	otherwise:	{BIT #1↑bit,α_0; BEQ lab}

As with several of the special cases treated earlier, the rationale is that the unary operators (TST, ROR, etc.) require one fewer words of code and one fewer memory references at execution time than the default BIT sequence.

Given the code generation functions described above we can now describe the operation of generating code for moving a field of one word to a field of another. In analyzing the kind of code one would like to produce for any particular such move it becomes apparent that various subcases arise in connection with

the position and size of the subfields of each of the words involved. These cases are:

full word:	p = 0, s = 16
byte:	p = 0 or p = 8, s = 8
bit:	p = anything, s = 1
other:	all other fields

The various cases of movement from one of these types of fields are treated separately by functions m_0-m_7 as specified by the following table:

source field type	destination field type: word	byte	bit	other
word	m_0	m_1	m_7	m_4
byte	m_2	m_1	m_7	m_4
bit	m_6	m_5	m_7	m_7
other	m_3	m_4	m_7	m_4

The functions are explained below:

m_0 (full-word to full word case):
  $z(e_2)$: {CLR α_1}
  otherwise: {MOV α_2,α_1}

m_1 (full word or byte to byte case):*
  $z(e_2)$: {CLRB α_1}
  otherwise: {MOVB α_2,α_1}

* Once again we are ignoring the complexities which arise due to: (1) the nonsymmetric properties of byte operators with respect to registers and memory, (2) word boundary problems with word instructions, and (3) the possibility that either $\alpha_1 = \alpha_2$ or that the evaluation of α_2 depends on α_1 (e.g. by indexing). These aspects increase by, roughly, fourfold the total number of cases to be considered.

m_2 (byte to full word cases):
{CLR α_1; BISB α_2,α_1}

m_3 (arbitrary field to full word):
{MOV α_2,α_1}; shift($e_1,p_2,0,s_2$); isolate($e_1,0,s_2$)

m_4 (arbitrary field to arbitrary field):
z(e_2): clearfield(e_1,p_1,s_1)
o(e_2): {BIS #(1↑s_1-1)*1↑p_1,α_1}
otherwise: clearfield(e_1,p_1,s_1); dial(e_2,p_1,s_1,p_2,s_2);
{BIS α_2,α_1}

m_5 (single bit to byte):
clearfield(e_1,p_1,8); l←gl; bittest(e_2,p_2,l);
{INCB α_1}; pl(l)

m_6 (single bit to full word):
clearfield(e_1,0,16); l←gl; bittest(e_2,p_2,l);
{INC α_1}; pl(l)

m_7 (move to/from arbitrary one bit field):
$\lambda(e_2) \wedge [\ (e_2) \bmod 2=1]$: {BIS #1↑$p_2,\alpha_1$}
$\lambda(e_2) \wedge [\ (e_2) \bmod 2=0]$: {BIC #1↑$p_2,\alpha_1$}
otherwise: clearfield(e_1,p_1,1); l←gl; bittest(e_2,p_2,l);
<u>if</u> p_1-0 <u>or</u> p_1=8
<u>then</u> {INCB α_1}
<u>else</u> {BIS #1↑p_2,α_1};
pl(l)

Some examples of various assignment expressions and the code they generate are given in Figure 30 .

5.2 Generating the <u>incr</u> Loop Code

In this section we wish to examine the generation of the code for the <u>incr</u> expression -- which is the Bliss counterpart of the Algol <u>for</u> statement or Fortran DO statement. The form of the <u>incr</u> expression is:

<u>incr</u> i <u>from</u> e_1 <u>to</u> e_2 <u>by</u> e_3 <u>do</u> e_4

Source Code	Object code (all constants are octal)	
X←.Y;	MOV	Y,X
X<0,8>←.Y<8,8>;	MOVB	Y+1,X
X<0,8>←0;	CLRB	X
X<5,3>←7;	BISB	#340,X
X<13,2>←.Y<3,2>;	MOVB	Y,R5
	SWAB	R5
	ASL	R5
	ASL	R5
	BIC	#117777,R5
	BIC	#60000,X
	BIS	R5,X
X<3,1>←.Y<4,1>;	BICB	#10,X
	BITB	#20,Y
	BEQ	L1
	BIS	#10,X
L1:		
X<0,8>←.Y<5,1>;	CLRB	X
	BITB	#40,X
	BEQ	L2
	INCB	X
L2:		
X<14,2>←.Y<0,2>;	MOVB	Y,R5
	ROR	R5
	ROR	R5
	ROR	R5
	BIC	#37777,R5
	BIC	#140000,X
	BIS	R5,X
X<7,9>←.Y<0,9>	MOV	Y,R3
	SWAB	R3
	RORB	R3
	ROR	R3
	BIC	#177,R3
	BIC	#177600,X
	BIS	R3,X

Figure 30. Code produced for some assignment expressions.

The semantics of Bliss specify that the counter variable, i, is local to the loop, and that it is to be initialized to the value of e_1. Then the expression e_4 is to be repeatedly evaluated (with i being increased by the value of e_3 after each such execution of e_4) so long as the value of i is not greater than e_2. The values of e_2 and e_3 are computed only once -- before e_4 is evaluated the first time. Associated with an incr expression are also the χ and ρ sets described in Section 2.5; the expressions in these sets must be evaluated prior to entering the loop body.

The "general case" code for the incr expression is shown below. Opportunities arise for optimizing this code when certain of e_1, e_2 and e_3 are constant:

```
        evaluate e1
        evaluate e2
        evaluate e3
        evaluate χ,ρ expressions
        store e1 into i
        branch to L2
L1:     evaluate e4 (the body)
        increment i by e3
L2:     compare i and e2 and conditional branch to L1
```

Consider, for example, the following loop:

incr I from .X to .Y by .Z do (.Y+3)←.(.Y+3)+1;

This loop produces the following code:

```
        MOV X,R4     ;Load e1 into i (R4 in this case).
        MOV Y,R5     ;Evaluate the χ set (".Y+3" in this case).
        ADD #3,R5
        BR  L3       ;Branch to the end condition test.
L2:     INC @R5      ;The body!
        ADD Z,R4     ;Increment the loop index.
L3:     CMP R4,Y     ;Compare and branch back if not done.
        BLE L2
```

In this particular case FLO noted that both the end condition,

".Y", and the step size, ".Z", were invariant in the loop; therefore neither was explicitly loaded into a temporary. This observation is purely the result of FLO and will not be discussed here.

Another interesting aspect of the algorithm and of this example should be noted. First notice that in the algorithm the expressions e_1, e_2, e_3, X, and ρ are computed before the value of e_1 is stored into the loop variable, i. Yet in the example code the first instruction loads the value of e_1 into register R4 (which is the location used for i in this case). Why? The explanation is simple, yet illustrates the care for detail one must exercise in such cases. In TNBIND distinct temporary names are created for e_1 and i, but they are placed in the same "preference class." Thus, unless there are a great many registers in use within the loop, the two temporary names will be bound to the same location by the PACK algorithm. If on the other hand there is a heavy demand for temporary locations, the two temporaries will compete on their individual merits for the available registers.

But another question remains -- why separate the formation of e_1 from its assignment to i? The answer to this question is rather more subtle and is related to the way in which the stack is managed and the relative cost of accessing the top element of the stack versus any other stack element. Since the top element of the stack is less expensive to access than the others, it is preferable to use this location for the index variable if no register is available. Furthermore, this is relatively easy to do since the body of a loop must not have any net effect on the depth of the stack. However, if the first use of i preceded the evaluation of e_1, e_3, etc., and any of these involved a function call with parameters, the desired effect would be lost. Consider a loop such as:

<u>incr</u> I <u>from</u> 1 <u>to</u> F(.X) <u>do</u> <body which uses 4 registers>

If the loop index is not used in the body, then the following code will be produced:

```
        MOV  X,-(SP)
        JSR  PC,F
        MOV  #1,@SP
L1:     <body which uses 4 registers>
        INC  @SP
```

```
        CMP  @SP,R0
        BLE  L1
```

However, if I is used in the loop, and hence is more important, the following will be produced:

```
        MOV  X,-(SP)
        JSR  PC,F
        MOV  R0,@SP
        MOV  #1,R0
L1:     <body which uses 4 registers>
        INC  R0
        CMP  R0,@SP
        BLE  L1
```

Now consider another loop similar to an earlier example, but with all of e_1, e_2, and e_3 being constants:

<u>incr</u> I <u>from</u> -10 <u>to</u> -1 <u>by</u> 1 <u>do</u> (.Y+3)←.(.Y+3)+1;

The code produced for this loop is:

```
        MOV  #177766,R4
        MOV  Y,R5
        ADD  #3,R5
L2:     INC  @R5
        INC  R4
        BLT  L2
```

Three things have happened. First, since both the beginning and ending values of the index variable are known at compile-time, and in particular since the loop body will execute at least once, the branch to the conditional test preceding the first iteration is eliminated. Second, since the data manipulation instructions set the condition codes, the comparison itself can often be avoided. Third, the identity $(i \leq -1) \equiv (i < 0)$ was exploited, thus the condition code settings resulting from the "INC R4" were used directly.

In order to fully explain the code generation for the <u>incr</u> expression it is convenient to introduce a few more functions and predicates:

(1) notecode — Returns a unique identification of the most recent instruction generated; used in conjunction with "anycodesince" below.

(2) anycodesince(c) — The parameter must be the value returned by some previous call on "notecode." The value of "anycodesince" will be true iff any code has been generated since the corresponding call on notecode.

(3) genmove(e_1,e_2) — Generates code to move the result of the expression e_1 to the location named by the expression e_2; this is <u>precisely</u> the function described in the previous section.

The algorithm for generating the code for the <u>incr</u> expression is then:

```
l1←gl; l2←gl;
C(e1);
C(e2); c←notecode;
C(e3); C(χ); C(ρ);
genmove(e1,i);
if z(e2) ∧ λ(e1) ∧ anycodesince(c) then {TST i; BR l2} else
if   λ(e1) ∨ λ(e2) ∨ [ (e1) >  (e2)] then {BR l2};
pl(l1); C(e4);
{ADD α3,i};
pl(l2);
if z(e2) then {BLE l1} else
   if  λ(e2) ∧  (e2)=-1 then {BLT l1} else {CMP α2,i; BLE l1}
```

One of the cases covered by this algorithm was not covered by the previous discussion. Consider a loop whose end condition is zero but the starting point is not known at compile time; for example

<u>incr</u> I <u>from</u> .X <u>to</u> 0 <u>by</u> 1 <u>do</u> <body>

In this case the action of incrementing the index variable at the end of the loop will set the condition codes and therefore an explicit "TST" instruction is not needed when this path is

followed. However, before the first time the <body> is executed it is necessary to insure the condition codes are set properly. Therefore the following code is generated:

```
      MOV  X, R5
      <any X and ρ code>
      TST  R5*
      BR   L2
L1:   <body>
      INC  R5
L2:   BLE  L1
```

5.3 Generating Boolean Operations

Boolean connectives (and, or, not, ...) which determine the flow of control produce no code, but rather merely determine the target for the control flow determined by their operands. Hence in this section we are not concerned with such boolean connectives, but only with those which are needed to produce 'real' results (e.g., A←.B <u>and</u> .C). The routines in the code module dealing with booleans simply ignore the flow case, the remainder of this section will do likewise.

A dominating factor in the first part of this discussion is the fact that the PDP-11 does not have a logical <u>and</u> instruction. Rather the instructions available are:

BIS	a,b	(bit set)	$a \vee b \rightarrow b$	the logical "or"
BIC	a,b	(bit clear)	$\sim a \wedge b \rightarrow b$	the logical "and not"
COM	a	(complement)	$\sim a \rightarrow a$	the logical "not"

Although not explicitly discussed in the chapter on DELAY, the node-specific DELAY routines for the boolean operators postpone complement in much the same way as the add delayer postpones negation. Thus the code generators for the booleans must be prepared for their operands to be in either complemented or uncomplemented form. We can illustrate the various code possibilities with the two tables shown in Figure 31. In each of

---

* This instruction is generated only if any code intervened between the "MOV X,R5" and this point.

these we assume: (1) e_1 is the target path, (2) that the non-target, e_2, is destroyable, and (3) that the "true," uncomplemented, result is to be formed.

e_1 and e_2	e_2	$\sim e_2$
e_1	COM e_2 BIC e_2,e_1	BIC e_2,e_1
$\sim e_1$	COM e_2 BIS e_2,e_1 COM e_1	BIS e_2,e_1 COM e_1

e_1 or e_2	e_2	$\sim e_2$
e_1	BIS e_2,e_1	COM e_2 BIS e_2,e_1
$\sim e_1$	COM e_1 BIS e_2,e_1	COM e_2 BIC e_2,e_1 COM e_1

Figure 31. CODE tables for boolean operators.

Clearly these tables are simple applications of the axioms of boolean algebra and merely become larger to account for the other variations which result from relaxing the restrictions stated above. This type of straightforward analysis is not the reason for introducing this section; rather we want to consider another important "targeting" concept. Consider the following example:

A<2,2>←.B<1,2> <u>or</u> .C<0,2>

Once again the lack of general byte extraction and storing on the PDP-11 becomes a major concern. In order to minimize the amount of shifting done, one must be careful where the "or" operation is performed. In this particular case it is best to shift the value of C left by one, then form the "or," and finally shift the result to align it with the field in A. Thus the following code is generated:

```
MOV  @#C,R0
ASL  R0
BIS  @#B,R0
ASL  R0
BIC  #6,R0
BIC  #6,A
BIS  R0,A
```

In order to generate code such as this it is necessary to know the position and size of the ultimate destination of the value as well as those of the operands. Thus a similar appearing statement should produce quite different code:

A<0,2>←.B<1,2> <u>or</u> .C<0,2>

```
MOV  @#B,R0
ASR  R0
BIS  @#C,R0
BIC  #6,R0
BIC  #6,A
BIS  R0,A
```

In order that the requisite information is available, a "target" position and size is passed down by the familiar mechanism. In particular such information generally originates from store nodes, which pass the position and size attributes from their left hand side operand to their right hand side. Of course this position and size target is passed uniformly to all operators; we have merely chosen to illustrate it with booleans.

Let us return to the booleans to see how the position and size target information is used. There are two effects; one is on the shifting sequence, the other is on the placement of masking (field isolation) operations. We shall consider the issue of shifting first. In this disucssion we assume t_1, t_2, and t_0 are the temporaries associated with the target, non-target and result.

There are really only three interesting sequences of operations if we focus only on the shifting aspect of the problem:

1) Shift t_1 to align it with t_2
   perform the boolean operation
   shift t_0 to align it with the target.

2) Shift t_2 to align it with t_1
   perform the boolean operation
   shift t_0 to align it with the target.
3) Shift t_1 to align it with the target
   shift t_2 to align it with the target
   perform the boolean operation.

We need only determine the minimum cost of the three. To this end the shift routine described earlier has a companion which performs the same analysis, but only returns the number of instructions involved.

In the two previous examples the size fields were all identical. Had they differed and, in particular, if the size of the target field had exceeded one of the operands, it might have been necessary to perform field isolation earlier. For example,

A<2,5>←.B<1,4> <u>or</u> .C<0,2>

must produce

```
MOV  @#C,R0
ASL  R0
BIC  #6,R0
BIS  @#B,R0
ASL  R0
BIC  #176,R0
BIC  #174,A
BIS  R0,A
```

In general, if one operand field is smaller than either the other operand or the target, that field must be isolated before the operation is performed.

Chapter 6

FINAL

Ah, finally...

FINAL is the last phase of the compiler and has two responsibilities: (1) performing a collection of ad hoc optimizations on the code produced by CODE, and (2) preparation of the final listing and relocatable code. The latter aspect is irrelevant to the purpose of this book and will not be discussed. The optimizations performed at this stage all result from looking only at the object code and make no use of the more global context available in the tree representation. These optimizations include those which have been called "peephole" optimizations [McK65], but have been considerably extended.

It is reasonable to ask why further optimizations should be necessary at this stage, given that reasonable care has been taken in the earlier phases, especially such ad hoc, low level ones as will be described. There are three reasons. First, some information isn't known until the code is generated. We shall see several examples of this, but the code generator, for instance, will happily create a branch instruction whose destination is another branch. It is better, of course, for the first instruction to branch directly to its ultimate target. Second, adjacent instructions in the code stream may have been generated from widely distant points in the tree; what seemed like good local code at each of those points may not look so clever when juxtaposed. Third, the FINAL optimizations themselves create new adjacency relations which may then admit reapplication of one or more of the other optimizations.

6.1 The Final Data Structure

FINAL receives its input from the CODE module in the form of a doubly linked list. This data structure allows FINAL to add and/or delete instructions anywhere in the code stream and to scan the code in both the forward and reverse directions. The items on this list are of two types: code cells and label cells. Each code cell contains a description of a complete PDP-11 instruction. Each label cell marks a position in the object code addressed by an instruction in one of the code cells. Each label cell has a sublist consisting of items called reference cells. There is a unique reference cell for each instruction that refers to a given label, and that reference cell contains a back pointer to the instruction. Thus FINAL is able to examine sequences of instructions which are adjacent in either the sense of being statically juxtaposed or, more importantly, in the sense of the dynamic execution order. This data structure is illustrated in Figure 32.

FINAL performs optimizations on this data structure in three subphases. The first subphase examines the cells sequentially and attempts optimizations based on relationships between each code cell and its neighbors (both static and dynamic). Since successful optimizations may change these relationships, this first subphase is repeatedly applied until no more improvements are possible. The second subphase examines each code cell exactly once and makes those changes possible by examining the instruction in isolation from its neighbors. The third phase is required by the nature of branch instructions on the PDP-11. There is a short form (branch, BR) which is restricted to referencing instructions located within 128 words of the branch itself; there also is a long form (jump, JMP) which may reference anywhere in PDP-11 memory. Obviously Bliss could use only the JMP instruction everywhere, but this would increase the size and execution time of programs. Instead, the third subphase of FINAL uses the JMP only where absolutely necessary. Since the use of a JMP instead of BR depends on the size of the program while the program size depends on how many JMP's must be used, this third subphase is not trivial.

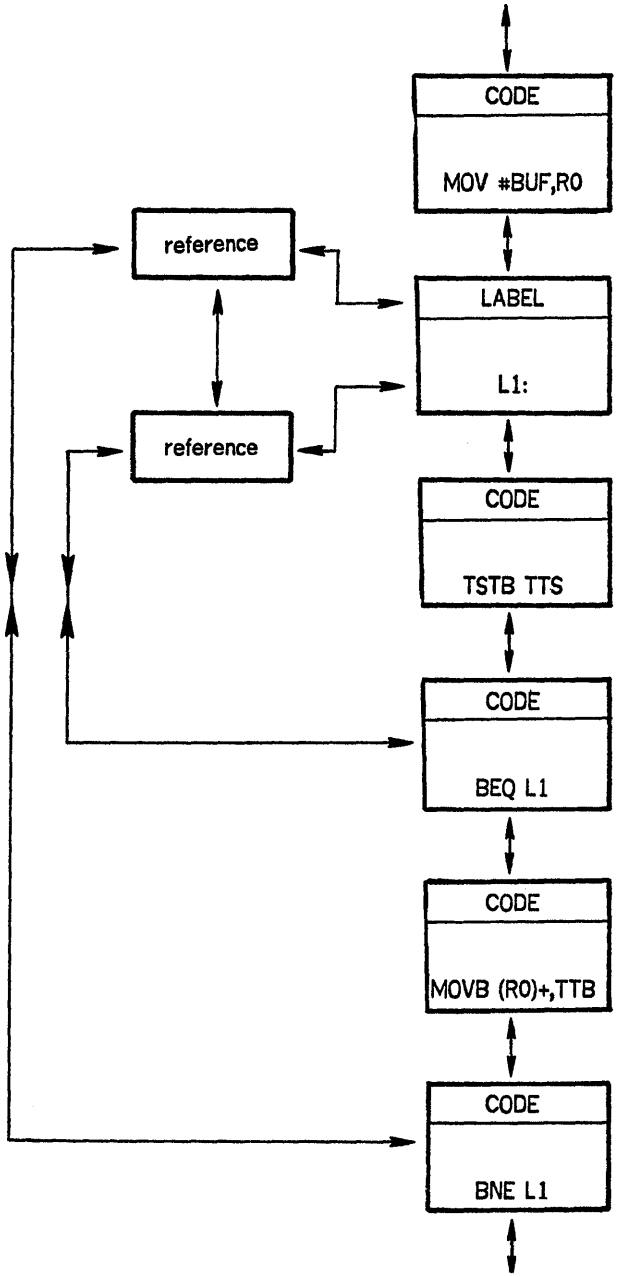


Figure 32. The FINAL data structure.

6.2 The First Subphase

The first subphase of FINAL examines each item in the code stream and calls an appropriate routine to perform optimizations for that type of item. If during this examination a change is made to the code stream, a flag is set and the subphase will be repeated. Only when a pass is completed in which no changes are made will FINAL progress to its second phase. The discussion below is keyed to the actions taken for the various types of cells encountered during one pass of the loop.

Label Cells. The first thing that is done when a label cell is encountered is to delete adjacent label labels and attach their reference cells to the remaining label cell.

If the set of reference cells for a particular label is empty, this label is not needed and is deleted. Otherwise the label marks a point where several execution paths will merge. FINAL uses this merging relationship to perform an optimization called "cross-jumping." "Cross-jumping" is performed by backing over the two merging sequences, comparing the ends of these two code sequences and, if identical sequences are found, replacing each code cell in one sequence by a branch to the corresponding cell in the other sequence. Later optimization will remove any unnecessary branch instruction which may have been introduced by this procedure.

Consider, for example, the following example:

if .x then x ← .y + .z else x ← .w + .z

which might produce the following code (before and after this portion of FINAL):

```
        Before Final                    After Cross-jumping
         BIT  #1,X                      BIT  #1,X
         BEQ  L1                        BEQ  L1
         MOV  Y,X        FINAL          MOV  Y,X
         ADD  Z,X        →              BR   L3
         BR   L2                        BR   L2
L1:      MOV  W,X              L1:      MOV  W,X
         ADD  Z,X              L3:      ADD  Z,X
L2:                            L2:
```

Unfortunately the cross-jumping optimization cannot be quite as simple as implied by this example. The nature of the PDP-11 runtime stack adds several complexities. First, merging code sequences must have the same stack depth; the CODE phase of Bliss enforces this by placing a "stack-adjust" instruction (a literal add to the stack pointer) at the end of one of the merging code sequences. This causes two otherwise identical sequences to have different final instructions. Similar differences between merging code sequences are caused by PDP-11 addressing modes that combine register modification with memory accesses (e.g., the use of "auto-decrement" for "pop"). To prevent these differences from stopping cross-jumping, stack-adjusting instructions (unless identical in both sequences) are not compared during the "back-over" procedure. Instead the effect of stack-adjusting instructions and stack-adjusting addressing modes is saved. Comparison is then done on the remaining instructions. Care must be taken to remember the differing stack depths when doing this comparison. Also, no labels may be backed-over unless the stack depths are identical. When no more instructions can be backed-over, new stack-adjust instructions are inserted to guarantee proper stack depth along all possible execution paths.

This optimization is one of those mentioned in the introduction as being impossible until the object code is produced. It is, to us, surprisingly effective.

When a label is found during the first subphase its reference cells are scanned for unconditional branch instructions that address the label. The "back-over" procedure is then executed on sequences preceding all possible pairs of these unconditional branches.

Unconditional Branch Instructions. The first thing done when optimizing an unconditional branch is to delete all code cells following it until the next label cell is reached, since any such code is unreachable. This will, for example, eliminate the redundant "BR L2" in the previous example. The label referenced by a branch is then compared with the label following the branch. If they are the same, the branch instruction is deleted. Otherwise, a "branch-chaining" procedure is executed. "Branch-chaining" consists of testing for a branch that addresses an internal label which precedes another branch which in turn may address a label which precedes another branch, etc. The entire

length of such a chain is searched and each branch in the chain is changed to use the same addressing as the last branch in the chain.

Finally, after the branch-chaining procedure, the "cross-jumping" optimization described above is applied to the code sequences preceding the branch, and the label referenced by the branch.

Conditional Branch Instructions. The conditional branch and its neighbors are examined to determine whether they can be deleted and/or modified; tests such as the following are performed:

1. The item following the conditional branch is checked to determine whether it is the label addressed by the branch. If this happens, the branch is deleted.
2. The item following the conditional branch is checked to see if it is an unconditional branch immediately followed by the label addressed by the conditional. In this case the unconditional branch is made into a conditional one with reversed sense, and the original conditional branch is deleted.
3. The next code cell (intervening label cells may now be ignored) is found and tested for being an unconditional branch to the same label; if it is, the conditional branch is deleted.

If the conditional branch is not deleted by one of the above criteria, "branch-chaining" is performed on it. "Branch-chaining" commencing from a conditional branch is different from that commencing from an unconditional one for two reasons. First, several different types of branches may be links of the chain. For example, a chain headed by a BEQ (branch equal instruction) may have links of BEQ, BPL, BLE, BGE, BLOS, BHIS, and BR, all of which will branch when the zero indicator is set. Second, in a conditional branch-chain only the first instruction in the chain may be modified.

After branch-chaining has been performed, the code cell following the ultimate target label is examined. If this instruction is also a conditional branch, but the opposite sense from the original (so that the branch will never be taken), the original

conditional branch is modified so that it will transfer one location beyond the target.

The utility of the optimizations of both conditional and unconditional branches arises from two sources: (1) the effect of the optimizations themselves sometimes creates the conditions in question, and (2) since the optimizations are desirable due to (1), and therefore present, the CODE module can be somewhat simpler if it need not worry about creating such conditions. To see the effect of these optimizations taken together, consider another (contrived) example:

<u>if</u> .x <u>then</u> x ← .y + .z + .w <u>else</u> (y ← .q; w ← .r; x ← .y + .z + .w)

Notice that ".y + .z + .w" is <u>not</u> a common subexpression or eligible for α-motion due to the assignments in the <u>else</u> branch; it <u>is</u> eligible for ω-motion -- a fact which we wish to ignore here for expository purposes. The code generated would then be:

```
      BIT  #1,X
      BEQ  L1
      MOV  Y,R0
      ADD  Z,R0
      ADD  W,R0
      MOV  R0,X
      BR   L2
L1:   MOV  Q,Y
      MOV  R,W
      MOV  Y,R0
      ADD  Z,R0
      ADD  W,R0
      MOV  R0,X
L2:
```

Although the optimizations will not happen in the order described below, for purposes of exposition we will describe them as though each is performed as a separate pass. First cross-jumping will produce

```
      BIT  #1,X
      BEQ  L1
      BR   L6
      BR   L5
      BR   L4
      BR   L3
      BR   L2
L1:   MOV  Q,Y
      MOV  R,W
L6:   MOV  Y,R0
L5:   ADD  Z,R0
L4:   ADD  W,R0
L3:   MOV  R0,X
L2:
```

Eliminating unreachable instructions (the branches) and then removing unreferenced labels will yield the following:

```
      BIT  #1,X
      BEQ  L1
      BR   L6
L1:   MOV  Q,Y
      MOV  R,W
L6:   MOV  Y,R0
      ADD  Z,R0
      ADD  W,R0
      MOV  R0,X
```

Finally, the "condition reversal" applied to the "BEQ" followed by removal of the redundant "BR" and the unreferenced label will produce:

```
      BIT  #1,X
      BNE  L6
      MOV  Q,Y
      MOV  R,W
L6:   MOV  Y,R0
      ADD  Z,R0
      ADD  W,R0
      MOV  R0,X
```

Although the example above is somewhat contrived, it does serve to illustrate that the action of some of the FINAL optimizations creates situations where others become operable.

Test and Compare Instructions. Test and compare instructions do not modify their operands, but are executed solely for their effect of setting condition codes. Since all other instructions set the condition codes, any test or compare instruction where the next code cell is neither a conditional branch nor the start of a branch-chain sequence leading to a conditional branch may be deleted.

Since most instructions set the indicators, it is possible that a test instruction is not necessary at all. The code cell preceding a test instruction is checked to see whether it operates on the same data as the test; if it sets the indicators properly the test is deleted. If the cell preceding the test is a label cell instead of a code cell then the same check is made with the instructions preceeding each branch addressing the label. Should any of these instructions properly set the indicators, the corresponding branch is made to refer to the instruction following the original test.

Literal Operands. Some machine operations, such as "add," "subtract," "or," and "nand," often have literal constants as one operand. When such an instruction is located, the literal constant may be one of a set of special cases. For example, an "add" of a literal zero can be deleted (provided the next instruction is not a conditional branch) or a "move" of a literal zero can be made into the shorter instruction "clear."

If the literal is not one of the special cases, the items following in the code stream are examined until a label, a branch instruction or an instruction accessing the data operated on by the original literal is found. If the item on which this scan terminates is another literal operation of the same type, then this literal is modified appropriately and the original instruction is deleted.

Auto-increment and Auto-decrement Address Modes. One of the special features of the PDP-11 architecture is the auto-increment and auto-decrement modes of addressing. These address modes allow a memory access indirectly through a register and at the same time modify the contents of the

register.* This is useful for stack manipulations and other scans through consecutive memory locations. FINAL is responsible for exploiting this feature of the hardware.

This optimization is attempted whenever a literal add or subtract to a register is found. FINAL scans both forward and backward, searching for operand addresses suitable for optimization. In the usual case the modifications required are uncomplicated; an operand address is changed to use auto-increment or auto-decrement mode and the original add or subtract instruction is altered. However, more complex changes are possible. It is often necessary to move a literal add or subtract to a different position in the code, and occasionally the literal instruction must be split into two or more separate instructions. The goal of this optimization is the reduction of code size. This occurs because auto-increment and auto-decrement modes require fewer words than some other modes. In addition it is possible that the literal in the add or subtract instruction will be changed to zero, which then allows the instruction to be deleted.

Addressing Optimization. The PDP-11 hardware recognizes eight different addressing modes. However, FINAL recognizes 38 different addressing constructs. These constructs are differentiated by such considerations as whether the register being used for indexing is the program counter, push-down stack pointer or some other general purpose register and whether or not the indexing base contains a reference to an own variable location. Some of these constructs are more desirable than others because they execute faster, or take less space, or are more amenable to future optimization. FINAL attempts to use what it considers the best form of addressing.

In particular, whenever FINAL is asked to optimize a "move" or "clear" instruction it considers the source and destination operands as containing identical data. The code stream following the instruction is scanned. Scanning stops if a label cell, unconditional branch instruction or an instruction that might modify either operand or might destroy the addressing of either operand (e.g., a register used in an index operation) is encountered. If either of the operands appears in a scanned instruction, that operand is replaced by the other if the resulting form is "better."

* See Appendix A for a more detailed description.

Redundant Store Optimization. One other optimization also occurs with "move" and "clear" instructions. In this case, a scan is made backwards from the data store until a branch or an instruction that might use the destination location is found. During this scan any instruction whose sole purpose is to modify the destination location of the data move is considered a "redundant store" and is deleted.

6.3 The Second Subphase

The above completes the description of FINAL's first subphase. When no more first subphase optimizations are possible, the second subphase begins; this subphase makes a single pass over the code stream and performs various bookkeeping operations, as well as modifying certain instructions to equivalent, simpler forms. Below is a partial list of the single instruction modifications performed during the second subphase:

Before	After
ADD #1,X	INC X
ADD #2,SP	TST (SP)+
ADD #4,SP	CMP (SP)+,(SP)+
MOV @R1,@0(R1)	MOV (R1)+,@-(R1)
BIT #6,6(R1)	BIT @PC,6(R1)

All of the above reduce the size of the instruction by one word. Also at this time branches to the RTS instruction are replaced by an RTS. To facilitate cross-jumping code sequences which merge at the RTS generated by CODE, such replacement was not done during the first subphase. In the main block and interrupt routines similar optimizations are performed for the HALT and RTI (return from interrupt) instructions.

The major bookkeeping function is to compute the length and relative location of each instruction. Unfortunately this is not yet completely possible. The PDP-11 branch instruction takes two forms; there is a one-word branch instruction, BR, which can only branch to instructions located within 128 words of itself, and there is a two-word branch instruction, JMP, which can transfer to any location in memory. Conditional branches are the short form; hence if the destination is more than 128 words away, the

conditional must be fabricated from the reverse conditional and a JMP. This requires a total of three words. As a result, the second subphase of FINAL assumes that all instructions will take the short form and then computes a minimum length; these minimum lengths are used to compute a minimum relative position for each instruction and label. This information is needed in the third subphase described below.

6.4 The Third Subphase

FINAL now enters its third subphase in order to resolve the length and relative position of each instruction. Every possible attempt is made to use the short "branch" form instead of the long "jump" form of the transfer instructions. If an instruction cannot transfer directly to the label without using the long form, an attempt is made to branch to some other branch instruction whose destination is the desired label. As a result, this third subphase will build branch-chains leading to a label; the reverse of the algorithm used in the first subphase (to remove branch-chains) is used. (For example a BEQ, branch equal, instruction is allowed to be chained to a BLE, branch less than or equal instruction.) This branch-chaining not only saves space but usually saves time, since a two-instruction branch-chain is faster than the three-word conditional jump.

This third subphase operates by examining only those instructions of unresolved length. By taking differences between the minimum possible locations of the instruction and the label it references, it is possible to compute the minimum possible distance between the instruction and label. This same distance is also computed between the original branch instruction and those other branch instructions referencing the label suitable for branch-chaining. If all these minimum distances are greater than 128, then the long form jump must be generated and the length of this instruction is resolved. As FINAL scans for those instructions of unresolved length it also updates the minimum locations of all instructions and labels. The code is repeatedly scanned until a pass is completed, during which no changes are made. The correct location of each instruction and label is its minimum location, and all instructions of unresolved length will take their short form and minimum length. It follows that this is safe

from the fact that all unresolved instructions between any still unresolved branch-label pair will take their short form. One more pass must still be made over the code stream, however, because although the lengths and locations of each instruction are resolved, there are still some branches that cannot reach their addressed label. These branches are found and they are inserted in the shortest possible branch-chain.

6.5 An Example

The previous material, especially that on the first subphase, was necessarily presented in piecemeal fashion. The optimizations were presented in terms of the actions taken for various specific instructions and circumstances. Unfortunately, such a presentation fails to show how the optimizations interact; the following example is presented as an attempt to illustrate these interactions.

The example program is shown in Figure 33. The routine EXAMPLE performs a non-recursive, symmetric tree walk, printing the "information" at each node of the tree. That is, the order is first to print all left descendants of any given node, then to print the information at the node itself, and lastly to print the right descendants. The tree is represented in three parallel vectors: LEFT, RIGHT, and INFO. LEFT and RIGHT contain the indices (times two because of byte addressing on the PDP-11) of the left and right descendant nodes of a given node -- or zero if there is none. An auxiliary stack is used to keep track of those nodes which have not yet been printed. This stack is represented by a vector, S, and a variable, SP, which contains the address of the current "top" element of the vector; note that this stack grows backwards, towards decreasing addresses, because of the pecularities of the PDP-11 addressing structure. Although it is not really necessary to understand the source text of this example for our present purpose, this is a fairly typical use of Bliss and illustrates how some of the features of the language are used in practice.

Figures 34 and 35 show, respectively, the code before and after FINAL has been applied. Before FINAL the code occupies 43 words; after FINAL it occupies 24 words. Thus, in this case at least, FINAL reduced the program size by more than 40

```
module EXAMPLE=
begin

! Macros to define a stack-type

macro
        DECLARESTACK(SZ) =own VEC S[SZ] ; local SP; bind STKSZ=SZ; $,
        INITSTACK=  SP←S[STKSZ*2] $,
        PUSH(Z)=    (SP←.SP-2; .SP←(Z)) $,
        POP(Z)=     (Z←..SP; SP←.SP+2; .Z) $;

! Generally useful definitions

structure VEC[I] = (.VEC+.I);
linkage R1 = bliss(register=1);

macro NOVALUE= .VREG $,
        REPEAT= while 1 do$;

! The example routine

own VEC LEFT:RIGHT:INFO[100];
external R1 PRINT;

routine EXAMPLE(Z)=
        begin
        local T; DECLARESTACK(100);
        INITSTACK; T←.Z; PUSH(0);
        REPEAT
                begin
                while .LEFT[.T] neq 0 do (PUSH(.T); T←.LEFT[.T]);
                PRINT(.INFO[.T]);
                while (T←.RIGHT[.T]) eql 0 do
                        if POP(T) neq 0
                                then PRINT(.INFO[.T])
                                else return NOVALUE;
                end;
        NOVALUE
        end;

end eludom
```

Figure 33. The example program.

percent. Of course this example was chosen explicitly to show the effect of FINAL, and the effect is not always this large.

```
EXAMPLE:
            JSR     R1,SAV3
            MOV     S+310,R3
            MOV     12(SP),R2
            ADD     #177776,R3
            CLR     @R3
L5:L6:      TST     LEFT(R2)
            BNE     L7
            BR      L8
L7:         ADD     #177776,R3
            MOV     R2,@R3
            MOV     LEFT(R2),R2
            BR      L6
L8:U2:      MOV     INFO(R2),R1
            JSR     PC,PRINT
L9:         MOV     RIGHT(R2),R2
            TST     R2
            BEQ     L10
            BR      L11
L10:        MOV     @R3,R2
            ADD     #2,R3
            TST     R2
            BNE     L12
            BR      L13
L12:        MOV     INFO(R2),R1
            JSR     PC,PRINT
            BR      L14
L13:        BR      L4
L14:        BR      L9
L11:U3:     BR      L5
L15:U1:L4:  RTS     PC
```

Figure 34. Code for example before FINAL.

Rather than try to explain all the actions of FINAL on this routine, we shall focus on those related to the later portion of the routine -- namely that portion of the routine beginning with the first PRINT(.INFO[.T]); and going through the end of the routine. The code originally produced for this portion begins with the line labeled "L8:U2:" in Figure 34.

Some of the actions of FINAL are obvious. For example,

```
EXAMPLE:
        JSR     R1,SAV3
        MOV     S+310,R3
        MOV     12(SP),R2
        CLR     -(R3)
L5:     TST     LEFT(R2)
        BEQ     U2
        MOV     R2,-(R3)
        MOV     LEFT(R2),R2
        BR      L5
U2:     MOV     INFO(R2),R1
        JSR     PC,PRINT
        MOV     RIGHT(R2),R2
        BNE     L5
        MOV     (R3)+,R2
        BNE     U2
        RTS     PC
```

Figure 35. Code for example after FINAL.

---

1. The redundant labels are removed.
2. TST instructions, such as that following L9, may be removed since the preceding instruction sets the condition codes properly.
3. Branch-chaining is performed so that, for example, the "BR L13" just before the line labeled L12 may be turned into "BR L4". Note that in this case this also removes the only reference to label L13, and that the label may also be removed.
4. Sequences such as

```
        BNE     L10
        BR      L11
L10:
```

   are converted into

```
        BEQ     L11
```

   In this case we even eliminate the label L10 since there are no more references to it.

5. The sequence "MOV @R3,R2; ADD #2,R3" may be altered to use auto-increment addressing.

If we apply those operations mentioned above it will be easier to see what happens next; we get:

```
U2:  MOV  INFO(R2),R1
     JSR  PC,PRINT
L9:  MOV  RIGHT(R2),R2
     BNE  L5
     MOV  (R3)+,R2
     TST  R2
     BEQ  U1
     MOV  INFO(R2),R1
     JSR  PC,PRINT
     BR   L9
     BR   U1
     BR   L9
     BR   L5
U1:  RTS  PC
```

Having done this, some new possible optimizations are exposed. In particular

6. The "TST R2" instruction can now be eliminated since the preceding instruction sets the condition codes properly.
7. The two instructions "MOV INFO(R2),R1" and "JSR PC,PRINT" preceding the "BR L9" may now be cross-jumped with the same two instructions preceeding the label L9.
8. The three branch instructions preceeding the final "RTS PC" are now unreachable by virtue of having had their labels removed.

Applying these optimizations we get:

```
U2:  MOV  INFO(R2),R1
     JSR  PC,PRINT
L9:  MOV  RIGHT(R2),R2
     BNE  L5
     MOV  (R3)+,R2
     BEQ  U1
     BR   U2
U1:  RTS  PC
```

Once again we have exposed another optimization -- namely:

9. The sequence of conditional branch and branch immediately preceding the RTS may be replaced by the reverse relational branch.

Applying this we obtain the final version:

```
U2:  MOV  INFO(R2),R1
     JSR  PC,PRINT
L9:  MOV  RIGHT(R2),R2
     BNE  L5
     MOV  (R3)+,R2
     BNE  U2
     RTS  PC
```

6.6 A Final Comment

The entire action of the FINAL phase may be viewed as a generalization of the "peephole" optimization described by McKeeman [McK65]. Alternatively, it may be viewed as a final polishing, or burnishing, of the code to remove rough edges left by the earlier phases of the compilation process. In some sense it simulates the behavior of the diligent assembly language coder attempting to get the last ounce of performance from his program. It is without doubt the most ad hoc, least formalized, and perhaps least aesthetically pleasing of the phases of the compiler. Yet it is one of the most effective. As we have noted earlier, all the fancy optimization in the world is not nearly as important as careful and thorough exploitation of the target machine; FINAL exists primarily to extend our exploitation of the target machine.

It is difficult to determine to what extent the FINAL optimizations would be needed if more complete algorithms, rather than heuristics, existed in earlier phases of the compiler. However, since some of the operations of FINAL exist simply because the requisite information does not exist earlier, we suspect that there will always be a role for a module similar to FINAL in compilers with optimization aspirations similar to those of Bliss/11. Thus we would hope that our objections to the ad hoc, unformalized nature of FINAL will be removed by future research and formalization of this important class of optimizations.

Chapter 7

CONCLUSION

As stated in the introduction, our goal in this paper has been to describe the various optimizations used in a specific, real compiler and to show how these optimizations interact to produce high quality code. We hope that a secondary effect of the exposition has been to pinpoint areas in which further research is needed. While no single aspect of the compiler has been described at the level of detail of the implementation itself, we expect that most knowledgeable systems programmers can infer the implementation from the material which is presented.

Perhaps the first question which should be addressed is, "What's general about this design?" That is, which aspects of this particular compiler design, for a particular language and particular machine, are worthwhile and applicable in other compilers for other languages and other machines? We feel that there are four major ones.

(a) First, the overall structure of the compiler -- that is, the assignment of function and order of application of the several phases. In any compiler design it is essential that the appropriate information be available at the point at which decisions must be made. Except in those cases where no suitable algorithm is known, e.g., determination of evaluation order, the structure of the compiler does extremely well at this.

(b) Second, the approach to global flow analysis. We, at least, find this approach to supply both the rigor and the intuitive cleanliness which leads to a straightforward and bug-free implementation.

(c) Third, the tree walks and notion of "targeting," or passing

information down the tree, in each phase. We find this to be an extremely powerful tool in each of the phases -- and in large measure responsible for the quality of the resultant code. (The cognoscenti will recognize the similarity of this mechanism to the information passed up and down an Algol '68 parse tree.)

(d) Fourth, TNBIND. The general strategy of assigning temporary names, measuring their lifetimes and importance, and then packing them into the available storage cells in order of importance seems completely general and has proved highly effective. A better lifetime characterization than the one described here seems desirable, but that appears quite feasible.

A second topic we would like to touch on briefly is that of the interaction of the several phases -- particularly those interactions which are not so evident when the phases are discussed separately. From a structural point of view, one would like to minimize the number of assumptions each of the phases makes about the others. If this goal were achieved it would be simple to make modifications in one of the modules (phases) which, so long as the modifications preserve the assumptions which other modules make about the one being modified, preserve the overall correctness of the compiler.

In one sense we have achieved this goal quite well, in another hardly at all. The crucial assumptions made by each phase are that the tree structure has remained intact and that the various fields contain legal values. Both of these assumptions are relatively easy to preserve, particularly since the definition of the structure and access to the tree, as well as those functions which transform it, are isolated. However, there are a far larger and more subtle set of assumptions which relate to the quality of the code produced. Most of the modules make assumptions about the "preferred" shape or content of the tree for subsequent phases. One such, for example, is the special way in which DELAY handles the expression "ϵ-2", where ϵ is destroyable and the expression occurs in an addressing context. For any value other than 2, DELAY would not demand the subtraction, but would utilize the indexing ability of the machine. But in this one special case it forces the subtraction to be done. Why? Because

it knows that after the code is generated FINAL will convert this to auto-decrement addressing. This is one of the more blatant examples of such assumptions and perhaps could have been done in another way, but it has the virtue of being easily explained. The challenge we leave for the next compiler writer is to see whether such assumptions can be eliminated while continuing to produce the same quality of code, for it is certainly advantageous to do so.

Next, we would like to focus on those areas of compiler optimization which lack adequate formalization and/or for which complete algorithms are not currently known. Our experience has shown that, in some intuitive sense, the size, complexity, and bug-proneness of each of the compiler phases is inversely proportional to the degree of formalization to which that area has been subjected.

In listing areas for further work there is a great temptation to order them by their "importance," as far as their effect on the quality of the resultant code is concerned. We have resisted that temptation. The fact is that all areas are important. Given our initial goal of producing code which is better than that produced by human programmers, a change of even a few percent is significant. Moreover, even a single extra instruction in a deeply nested loop may have an effect on execution time completely disproportionate with its static effect on program size. Hence, the following list is simply ordered by the phases in the Bliss/11 compiler within which the problem arises.

(1) DELAY

(a) Evaluation order. As noted in the text, Bliss/11 uses a heuristic extension of a well-known algorithm for choosing the evaluation order of expressions in order to minimize the number of in-use registers. A complete algorithm, or at least a better heuristic, which works in the presence of globally redundant expressions (including those involved in code motion optimizations) is badly needed.

(b) Desirable vs. feasible optimizations. This is currently done on a completely ad hoc basis, and requires a very detailed knowledge of the target machine. It is, in fact, an instance of a more general problem of deciding when to commit to an optimization.

(c) Exploiting addressing hardware. The mechanisms used in DELAY are critical to producing object code which exploits the target machine. We believe that these ideas can be generalized and expressed in a machine-independent way.

(2) TNBIND

(a) Targeting and preferencing are ideas which we have not seen in the literature before. They have a significant effect in Bliss/11, and we believe they can be generalized in a machine-independent way.

(b) Floating temporaries. Bliss/11, like most compilers, dedicates a location (usually a register) to hold the value of a temporary (or user variable) over its entire lifetime. There are many contexts in which this is non-optimal, but we know of no computationally reasonable formulation which avoids this commitment.

(c) Lifetime characterization. The lon/fon characterization of the lifetime of a temporary is relatively inexpensive, but incomplete. A scheme is needed which is both computationally reasonable and (more nearly) complete.

(3) CODE

When all is said and done, the quality of highly local code is absolutely essential to the overall performance of the object program produced by a compiler. No matter how clever the other phases of the compiler, if the local code is sloppy, the compiler will be a disaster. Because of the highly machine-specific nature of CODE it is possibly inevitable that the construction of this phase will be relatively ad hoc. However, it is not the ad hoc nature of the implementation of this phase which disturbs us, but rather the generation of the special cases which should be considered by the implementation. The set of cases treated in the text for general data moves, for example, represents a substantial investment of intellectual effort, has been modified many times as new cases were uncovered, and still has no guarantee of being exhaustive. It seems reasonable to us that the generation of these cases can be mechanized.

(4) FINAL

All of FINAL is ad hoc, yet its effect is significant. It is very hard to determine whether FINAL would be so important if more complete algorithms existed at earlier stages of the compilation process; nevertheless, the actual implementation is highly regular -- suggesting to the authors that its effects could be both formalized and extended. It is also difficult to avoid drawing analogies between certain aspects of FLO and FINAL (notably cross-jumping); we would like to believe that the analogy is more than accidental.

We would like to conclude by presenting our view on the importance of efficiency. It is fashionable in some circles, especially some academic ones, to deprecate the importance of efficiency. The argument goes: "It is programming time, not execution time, that matters. What difference does it make if the program runs one minute or two?" This argument is often used, for example, to encourage the use of some of the newer, often less efficient, programming languages.

It is difficult to argue with this. In fact we don't for our own programs. Our time is <u>much</u> more important than the machine's. However, the other fellow's program (yours) is an entirely different matter! Every second that his program executes, ours can't. If his program is inefficient, we are the ones who suffer, not him. The problem of efficiency is <u>not</u> one of how long it takes to run <u>a</u> program, but rather one of obtaining the maximum benefit from a finite resource.

The reason that compiler optimization is important is that programmer efficiency and execution efficiency need not be a choice we must make. Optimization is a technological device to let us have our cake and eat it, too -- to have <u>both</u> convenient and well-structured programming and efficient programs.

APPENDIX A

PRIMER ON THE PDP-11

In order to understand and appreciate some of the optimizations described in this paper some understanding of the target machine, the PDP-11, is necessary. This appendix is not a definitive description of the PDP-11. It is, rather, a brief description of some of the addressing features and enough of the instruction set to allow the reader to understand the examples; in particular the i/o and interrupt structure of the machine are totally ignored.

Basic Properties

The PDP-11 is a "minicomputer" with a word size of 16 bits; memory is byte (8 bits) addressable. Because a memory address is constrained to exist in a single word, directly addressable memory is limited to 2^{16} (64K) bytes or 2^{15} (32K) words. The processor contains eight "general purpose" registers (R0-R7), two of which are assigned specific uses (R6 = SP and is used as a stack pointer for subroutine calls, and R7 = PC and is the program counter). A ninth register in the processor -- called the "program status register," PS -- contains a set of condition code bits which are set by the action of data manipulation instructions. The condition code bits are

Z = 1 iff the result of the operation was zero
N = 1 iff the result of the operation was negative
C = 1 iff the result of the operation produced a carry out of its most significant bit
V = 1 iff the result of the operation produced an arithmetic overflow

The data manipulation instructions of the PDP-11 are classified as either unary (monadic) or binary (diadic). In both cases the

instructions may be used as zero address (i.e., stack oriented with reverse polish operators) or single address (conventional general register) architecture. Binary (diadic) instructions may, in addition, be viewed as two address (memory-to-memory).

Addressing Structure

In this section we shall use one of the binary operators to illustrate the addressing options, namely "MOV src, dst" which moves a word from the location specified by the source address, "src", to the location specified by the destination address, "dst".

The group of binary instructions are encoded in a 16-bit word as shown below.

OPERATOR (4 bits)	SOURCE (6 bits)	DESTINATION (6 bits)

The source and destination fields are coded identically, as shown below. As should be obvious, these six-bit fields are not adequate to hold memory addresses; rather some encodings of these fields specify that words following the instruction contain operands or addresses.

mode	d	reg

The "reg" field always specifies one of the eight processor registers. The "d" bit, when set, specifies one level of indirection or "deferral." The meanings of the mode field values are given below for d = 0 (one level of indirection should be added to these descriptions for d = 1).

m=0 register mode
The operand to be used in the instruction is contained in the register specified by the "reg" field.

m=1 auto-increment mode
The register specified by the "reg" field contains the

address of the operand to be used. After being used, the specified register is incremented. (The register is incremented by either one or two depending upon the nature of the instruction -- "byte" instructions increment the register by one and "word" instructions increment the register by two.)

m=2 auto-decrement mode

The register specified by the "reg" field specifies the address of the operand to be used. However, before being used the value of the register is automatically decremented by either one or two as in the m=1 case.

m=3 index mode

The register specified by the "reg" field is used as an index register and is added to the 16-bit word following the instruction in order to obtain the address of the operand to be used in the instruction. The PC is automatically incremented (by 2) to point beyond the index word.

The following notations, used by the PDP-11 assembler, will be used to illustrate the wide range of addressing constructs possible on the PDP-11:*

r	Register mode (m=0, d=0).
@r	Indirect register mode (m=0, d=1).
(r)+	Auto-increment mode (m=1, d=0).
@(r)+	Indirect auto-increment mode.
-(r)	Auto-decrement mode.
@-(r)	Indirect auto decrement mode.
x(r)	Indexed mode.
@x(r)	Indirect indexed mode.

Three special cases involving the use of the program counter as the register named in an address specification are worthy of special note:

(1) literal operands

The notation "#x" denotes (PC)+ addressing where the literal, "x", has been placed in the word following the

* In this table the symbol "r" is used to denote any of the eight processor registers and "x" denotes a constant.

instruction; the effect is to use the literal "x" itself as the operand.

(2) relative addressing

The location of a data item may be expressed as a displacement of the current instruction. This is the preferred method of accessing data. Hence a symbol, e.g., "A", is generally interpreted as "α(PC)" addressing where α is some appropriate constant. Deferred relative addressing, "@A", is of course possible.

(3) absolute addressing

In a few cases relative addressing, as in (2), is undesirable; in these cases absolute addressing, denoted @#A, is possible. Absolute addressing is actually "@(PC)+" mode.

The following list illustrates some of the variety possible with the single MOV instruction given the variety of addressing modes; in the center column of this list, memory is abbreviated to "m", register to "r", and literal to "l". The lengths (in words) of the various forms are also given.

Instruction	Effect	Length
MOV r0,r1	r-to-r	(1)
MOV r0,X	r-to-m	(2)
MOV X,r0	m-to-r	(2)
MOV 5(r0),r1	m (indexed)-to-r	(2)
MOV r0,5(r1)	r-to-m (indexed)	(2)
MOV X,Y	m-to-m	(3)
MOV 4(r1),-2(r0)	m (indexed)-to-m (indexed)	(3)
MOV #17,r3	I-to-r	(2)
MOV #17,X	I-to-m	(3)
MOV #17,5(r2)	I-to-m (indexed)	(3)
MOV r1,-(SP)	push r contents onto stack*	(1)
MOV (SP)+,r0	pop stack contents into r*	(1)
MOV #6,-(SP)	push I value onto the stack*	(2)
MOV X,-(SP)	push m contents onto the stack*	(2)
MOV (SP)+,X	pop stack contents into m	(2)

Of course, many more combinations are possible; these are sufficient to illustrate the flexibility of the addressing mechanism and to illustrate the various optimizations.

The Instruction Set

In this section we shall enumerate the PDP-11 instructions needed for the examples in the text. There are four instruction groups in the PDP-11 of interest to us, each with its own instruction format.

Binary Group:	op source destination
Unary Group:	op destination
Branch Group:	op offset
Jump Group:	op reg, <u>or</u> op destination

*Note that by convention the PDP-11 stack grows "down" toward lower addresses thus, for example, a "push" operation involves decrementing the stack r.

Binary Group

The "binary group" instructions specify two addresses -- a source address and a destination address. In general these instructions perform an operation between source and destination operands and place the result in the destination. The relevant instructions are:

MOV*	Move source to destination.
ADD	Add source to destination.
SUB	Subtract source from destination.
CMP*	Compare source and destination; set condition codes; no operands modified.
BIS*	"Bit set," form inclusive or of source and destination in destination.
BIC*	"Bit clear," form (~src ∧ dst) in destination.
BIT*	"Bit test," form and of source and destination; set condition codes; no operands modified.

Unary Group

The "unary group" instructions specify a single address -- the destination. In general these instructions perform an operation on the destination operand and return it to the destination location. The relevant instructions are:

CLR**	Clear destination word to zero.
COM**	Complement the destination operand.
NEG**	Negate the destination operand.
INC**	Increment the destination operand by one.
DEC**	Decrement the destination operand by one.
TST**	Compare the destination operand to zero; set the condition codes.
ROR**	Rotate right.
ROL**	Rotate left.
ASR**	Arithmetic shift right.
ASL**	Arithmetic shift left.

*These instructions also have a "byte form," e.g., MOVB which moves a byte (8 bits) from source to destination.

**These instructions also have a "byte form," e.g., CLRB which clears a single byte.

SWAB Swap the two bytes of the destination (word) operand.

It should be noted that all the shift instructions (ROR, ROL, ASR, ASL) are single bit shifts of the specified type; also, the rotate instructions involve 17 (not 16) bits -- the "carry bit," C, of the condition codes participates in these shifts.

Branch Group Instructions

These instructions branch conditionally on the setting of the condition codes. The instructions do not specify the address of the branch, but rather an offset from their own location (i.e., the value of PC). Since the offset is eight bits and denotes a word displacement, branches are limited to -128 or +127 words from their own location. The relevant instructions are:

BR	Branch (unconditional).
BEQ	Branch on equal (to zero).
BNE	Branch on not equal.
BMI	Branch on minus.
BPL	Branch on plus.
BLT	Branch on "less than."
BGE	Branch on "greater or equal."
BLE	Branch on "less than or equal."
BGT	Branch on greater than.
BHI	Branch on higher.
BLOS	Branch on lower or same.
BLO	Branch on lower.
BHIS	Branch on higher or same.

Two things should be noted about these instructions. First, all relationals (e.g., BLE) are comparisons against zero. However, since the compare instruction, CMP, performs an implicit subtract, a sequence such as "CMP A,B; BLE LABEL" corresponds to a branch on the relative magnitudes of A and B. Second, the branches BHI, BLO, etc. are termed "unsigned comparisons" since they treat the items as unsigned (positive) 16-bit numbers.

Jump Group Instructions

This group includes three instructions: (1) the unconditional jump, JMP, (2) the subroutine call instruction, JSR, and (3) the subroutine return instruction, RTS.

The JMP and JSR instructions specify normal addressing and thus are not constrained as to the distance which may be jumped, as the branch instructions are.

The JSR and RTS instructions are more involved than is necessary for our purposes. The only forms used are "JSR PC,SUBR" and "RTS PC" which together have the effect of transferring control to a subroutine leaving the return address on the stack, and of returning from a subroutine:

JMP	Jump.
JSR	Jump to subroutine.
RTS	Return from subroutine.

APPENDIX B

A SHORT PRIMER ON BLISS

The Bliss language is used throughout this book for examples. This appendix has been included for those who may not be familiar with the language. This primer is not a complete description of Bliss; it is, rather, an introduction to those aspects of the language used in the examples. More complete information on the language may be found in [Wul71, Wul72a].

Bliss is a descendant of Algol 60 in that it has block structure, a similar expression format (including operator precedence), similar control constructs, and potentially recursive procedures (called "routines"). It differs from Algol in its interpretation of identifiers, the omission of a goto, and the fact that it is expression-oriented rather than statement-oriented. Because of its resemblance to Algol, this primer will concentrate on the differences.

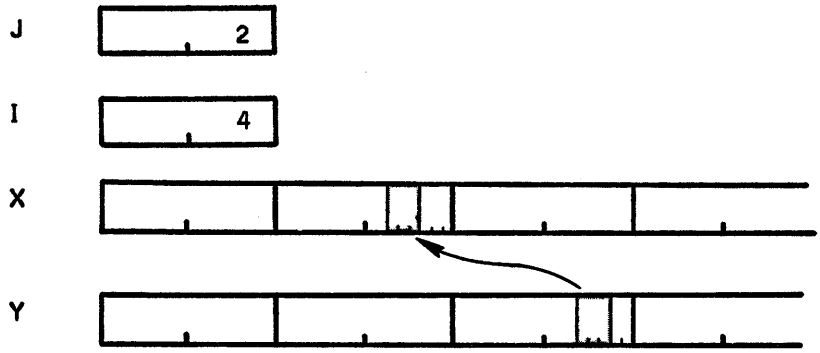
Interpretation of Names

A Bliss program operates with and on a number of storage "segments." A segment consists of a fixed and finite number of "words." A word may be "named"; the value of a name is called a "pointer" to the word. Identifiers are bound to names by declarations. Thus the value of an instance of an identifier, say x, is not the value of the word named by x, but rather a pointer to x. This interpretation requires a "contents of" operator for which the symbol "." has been chosen.

This context-independent interpretation of identifiers as pointers is maintained consistently throughout the language. It is the operators of Bliss which place an interpretation on the value of an expression. So, for example, the assignment operator "←" interprets its right hand operand as a value which is to be stored

in the word pointed to by the value of the left hand operand. As a result the effect of the Algol assignment statement "A:=B+C" is identical to the Bliss assignment "A←.B+.C". This interpretation of names also allows the computation of pointers in Bliss so that the effect of the assignment "(A+3)←.(B+5)" is to store the value of the fifth location beyond B into the third location beyond A.

While, as mentioned above, names are pointers, the converse is not true. Pointers are a more general concept and, in particular, may refer to a subfield of a word. The notation "ε<p,s>" is used to construct a pointer to a field within a word whose address is given by ε (an expression whose value may not be known until execution time), where the field is s bits wide and begins p bits from the low-order end of the word. Thus the expression X<0,3>←.Y<5,3> moves a 3-bit field beginning at the sixth bit of the word at location Y into the 3-bit field at the low-order end of X. More generally, the effect of (X+.J)<2,3>←.(Y+.I)<3,3> is shown in the diagram below.



Data structures

(The material in this section is not needed for the examples in the text and is only used in the examples in the appendicies. It may be skipped on first reading.)

Bliss takes a rather different view of data structures from that found in most programming languages. Rather than define data structures by their storage layout, Bliss defines them in terms of the algorithm used to access an element of the structure. Moreover Bliss allows -- in fact demands -- that the user define these algorithms; there are no "built-in" data structures. The following example illustrates the mechanism:

```
structure bytevector[i] = (.bytevector+.i)<0,8>;
own bytevector a[10];
a[3]←0;
```

The structure declaration defines the accessing algorithm; that is, given the base address of a particular bytevector and a particular index value, it specifies how to compute the address of the appropriate element. The own declaration serves both to allocate storage and to associate an access algorithm name with the identifier "a". The subsequent use of "a[3]" invokes the access algorithm associated with the name "a" to compute the appropriate address.

The syntax of the structure declaration was intentionally chosen to be similar to that of a subroutine, and the reader is encouraged to think of it as such. In practice, however, structures are expanded "in-line" to avoid linkage overhead, and so that appropriate special case optimizations may be performed.

Control Structures

Bliss is a block-structured, go-to-less, "expression language." That is, every executable construct, including those which manifest control, is an expression and computes a value. Expressions may be concatenated with semicolons to form expression sequences. An expression sequence is evaluated in strictly left-to-right order and its value is that of its last (rightmost) component expression. A pair of symbols, begin and end, or left-and-right parentheses, may be used to embrace such an expression sequence to form a simple expression. A block is a special case of the construction which contains declarations.

Other than expressions and functions, control mechanisms in Bliss fall into four classes: conditional, selection, looping, and leave. The conditional expression

$$\underline{\text{if}}\ e_0\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2$$

is defined to have the value e_1 just in the case that e_0 evaluates to true (a one in the low order bit) and e_2 otherwise. The abbreviated form "if e_0 then e_1" is considered to be "if e_0 then e_1 else 0."

The conditional expression provides two-way branching. The selection expressions, case and select, provide n-way branching:

case e_0 of set e_1; e_2; ...; e_n tes

select e_0 of nset e_1: e_2; ...; e_{2n-1}: e_{2n} tesn

The case expression is executed as follows: (1) the expression e_0 is evaluated, (2) the value of e_0 is used as an index to choose one of the e_j's ($0 \le j \le n$). The value of e_0 is constrained to lie in the range $0 \le e_0 \le n$. The value of the case expression is e_i ($i=e_0$). The select expression is similar to the case expression except that e_0 is used in conjunction with the e_{2j-1}'s to choose among the e_{2j}'s. The execution of the select expression above is described by the following equivalent Bliss expression:

```
begin
t←e;
v←-1;
if e1 eql .t then v←e2;
...
if e2n-1 eql .t then v←e2n;
.v
end
```

Hence the value of the select expression is that of the last e_{2j} to be executed, or -1 if none of them is executed.

The loop expressions imply repeated execution (possibly zero times) of an expression until a specific condition is satisfied. There are several forms, two of which are:

do e_0 while e_1

incr <id> from e_0 to e_1 by e_2 do e_3

In the first form, the expression e_0 is repeated as long as e_1 satisfies the Bliss definition of true. The second form is similar to the "step ... until" construct of Algol, except (1) the control variable, <id>, is implicitly declared to be local to the incr expression, and (2) e_0, e_1, and e_2 are evaluated only once (before

the evaluation of the loop body, e_3). Except for the possibility of a leave expression within e_3 (see below) the value of a loop expression is uniformly taken to be -1.

The control mechanisms described above are either similar to, or slight generalizations of constructs in many other languages. Of themselves they do not remove the inconveniences generated by removing the goto. Another mechanism is desirable -- the leave mechanism. A leave is a highly structured form of forward branch which is constrained to terminate coincidentally with some control environment in which the leave is nested. The general form is:

leave <label> with <expression>.

where <label> must be attached to a control environment within which the leave expression is nested. A leave expression causes control to immediately exit from a specified control environment. The <expression> defines the value of the environment.

Subprograms

Subprograms, called routines, are defined and called in Bliss much as in Algol. The following differences should be noted however:

(1) There is no specification part, and all parameters are implicitly call-by-value.
(2) The value of a routine is that of the expression which is its body, unless a return is executed.
(3) The return expression is equivalent to a leave except that it implicitly leaves the body of the routine.
(4) In order to call a routine which has no actual parameters, an explicitly empty actual parameter list must be used (e.g., "f()").
(5) It is possible, through a linkage declaration, to specify the parameter and calling-sequence conventions for individual routines.

APPENDIX C

A COMPLETE EXAMPLE

In this Appendix we will attempt to present a complete example -- that is, to take a short Bliss program and to show the effects of some of the various optimizations in the order in which they are applied. There are inherent difficulties in this attempt: in order to keep the example to tolerable length the program must be short; in order to illustrate several optimizations it must be fairly rich. The result of these two considerations is a contrived program. We must also lie a bit in the presentation in order to avoid tedious detail.

The example we shall consider is the simple program shown below:

```
begin
external f,g;
structure bytearray[i,j] = [i*j](.bytearray+(.i-1)*j-1+.j)<0,8>;
own bytearray a[8,8];

decr i from 8 to 1 do
    decr j from 8 to 1 do
        if .i eql .j then a[.i,.j]←0 else
        if .i gtr .j then a[.i,j]←f(.i+.j) else a[.i,.j]←g(.i+.j);
end;
```

If one cares -- and it certainly isn't essential to an understanding of the following material -- the effect of this program is to initialize a two-dimensional, Fortran-style (1 origin indexing) array. The upper triangular elements are initialized so that $a_{ij} = f(i+j)$; the lower triangular elements are initialized so that $a_{ij} = g(i+j)$; and the diagonal elements are set to zero. Thus the resulting matrix is shown in Figure C-1.

	1	2	3	4	5	6	7	8
1	0	f(3)	f(4)	f(5)	f(6)	f(7)	f(8)	f(9)
2	g(3)	0	f(5)	f(6)	f(7)	f(8)	f(9)	f(10)
3	g(4)	g(5)	0	f(7)	f(8)	f(9)	f(10)	f(11)
4	g(5)	g(6)	g(7)	0	f(9)	f(10)	f(11)	f(12)
5	g(6)	g(7)	g(8)	g(9)	0	f(11)	f(12)	f(13)
6	g(7)	g(8)	g(9)	g(10)	g(11)	0	f(13)	f(14)
7	g(8)	g(9)	g(10)	g(11)	g(12)	g(13)	0	f(15)
8	g(9)	g(10)	g(11)	g(12)	g(13)	g(14)	g(15)	0

Figure C-1. Array produced by sample program.

Since we shall be primarily concerned with the tree representation of the program, it may be simpler to think in terms of the program as it would appear after the structure accesses have been expanded and defaults supplied, namely:

```
begin
external f,g;
own a[64];

decr i from 8 to 1 by 1 do
   decr j from 8 to 1 by 1 do
      if .i eql .j then (a+(.i-1)*8-1+.j)<0,8>←0 else
      if .i gtr .j
         then (a+(.i-1)*8-1+.j)<0,8>←f(.i+.j)
         else (a+(.i-1)*8-1+.j)<0,8>←g(.i+.j);
end;
```

The final code produced for this example is given in Figure C-2; the purpose of this appendix is to demonstrate how this code is generated.

Figure C-3 shows the form of display we shall use to illustrate the tree representation of this program. Indentation is used to denote levels in the tree; vertical lines connect subnodes of a given node. The numbers along the left edge of the diagram identify the nodes. The symbol "ψ" is used to denote the identical subtrees formed by the structure accesses. Subsequent figures will show various types of information in the tree nodes; in order to avoid cluttering up the diagrams with too much detail, no one figure contains all possible information.

Figures C-4a through C-4d illustrate the tree after LEXSYNFLO but before delay. In particular, the CSTHREAD, CSPARENT, etc. fields have been attached to those nodes for which they are interesting. (A field is "interesting" here if it doesn't name itself or contain a null value.) In particular notice that

1. The subtree corresponding to "a+(.i-1)*8-1" has been recognized as constant in the inner loop (that is, the node corresponding to this subtree is in the χ-list attached to the inner loop).
2. The subtree corresponding to ".i+.j", which is a routine call parameter on both forks of the last <u>if</u> expression, has been recognized as a candidate for α-motion and has been attached to the α-list of that <u>if</u> node.

In this simple case, no other common subexpressions or interesting feasible optimizations were detected by FLO.

```
      MOV   #10,R1      ; initialize outer loop index in R1
L1:   MOV   R1,R5
      ASL   R5
      ASL   R5
      ASL   R5
      ADD   #A-9,R5     ; form (A + (.i-1)*8-1) in R5
L3:   MOV   #10,R3      ; initialize inner loop index in R3
L5:   MOV   R5,R4
      ADD   R3,R4       ; form entire address (A + (.i-1)*8-1+.j) in R4
      CMP   R1,R3
      BNE   L8          ; branch if .i ≠ .j
      CLRB  @R4         ; zero the diagonal element
      BR    L9
L8:   MOV   R1,R2
      ADD   R3,R2       ; form (.i+.j) outside the if expression
      CMP   R1,R3
      BLE   L13         ; branch if .i ≤ .j
      MOV   R2,-(SP)    ; push the argument and call "f"
      JSR   PC,F
      BR    L15         ; branch to end of conditional
L13:  MOV   R2,-(SP)    ; push the argument and call "g"
      JSR   PC,G
L15:  MOVB  R0,@R4      ; perform assignment for either branch
      TST   (SP)+       ; restore stack state at end of loop
L9:   DEC   R3          ; decrement inner loop index
      BGT   L5          ; loop if j ≥ 1
      DEC   R1          ; decrement outer loop index
      BGT   L1          ; loop if i ≥ 1
```

Figure C-2. Final version of code with annotations.

Figure C-5 illustrates the subtree for the expression "(a+(.i-1)*8-1+.j)" after DELAY but before TNBIND (most of the interesting effects of DELAY occur in this subtree). Note the following obvious changes:

(1) In i(4) the multiplication by eight has been changed to a shift by three.
(2) In i(3) the evaluation order of the operands has been reversed (by reversing the subnodes themselves). This will reduce the number of registers needed to compute "a+(.i-1)↑3".

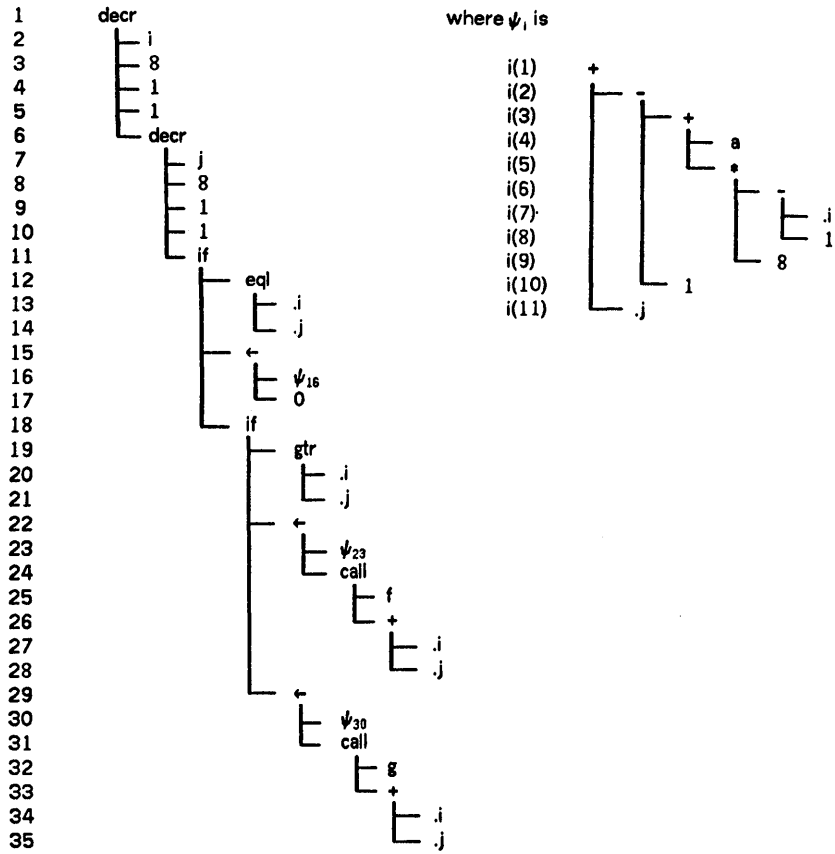


Figure C-3. Tree representation of the example.

Less obvious, perhaps, is the fact that the field settings shown in C-5 have effectively converted the expression

```
a+(.i-1)*8-1
into
(a-9)+.i↑3
```

Figure C-6 shows (partially) the temp name assignments and lon/fon values resulting from TLA (the lon/fon of "uninteresting" nodes have been deleted to emphasize those of greater importance). Figure C-7 illustrates the rectangular regions in the lon/fon space which characterize the "lifetime" of the various temporary names. In this particular case the lifetimes all overlap and hence the diagram is a bit uninteresting -- all the temporary names are assigned to different registers. Packing is also rather uninteresting since all the temporaries may be bound to registers.

Figure C-8 illustrates the code as generated by CODE. Several things should be noted:

1. Since the loop bounds are fixed, no code has been emitted to check them on the first time through.
2. The loop termination conditions, e.g., ".i < 1", have been modified to compare against zero; this is always more efficient on the PDP-11, but especially so when the condition codes have been implicitly set by an arithmetic operation.
3. Stores of zero have been performed by the more efficient CLRB instruction.

Comparison of Figures C-2 and C-8 shows the effect of FINAL; in particular note:

1. The sense of some conditionals has been reversed to save a BR instruction.
2. Operations involving constant arithmetic have been modified. In particular, "ADD #2,SP" has been modified to "TST (SP)+" and "SUB #1,R" has been changed to "DEC R" in two places.
3. Cross-jumping has eliminated one "MOVB R0,@R4" instruction.

```
1   decr
2   ├─ i
3   ├─ 8
4   ├─ 1
5   ├─ 1
6   └─ decr (X = <16(2)>)
7      ├─ j
8      ├─ 8
9      ├─ 1
10     ├─ 1
11     └─ if (α = 16)
12        ├─ eql
13        │  ├─ .i
14        │  └─ .j
15        ├─ ←
16        │  ├─ ψ16
17        │  └─ 0
18        └─ if (α = <26>)
19           ├─ gtr
20           │  ├─ .i
21           │  └─ .j
22           ├─ ←
23           │  ├─ ψ23
24           │  └─ call
25           │     ├─ f
26           │     └─ + (FPARENT=26, FSTHREAD=33)
27           │        ├─ .i
28           │        └─ .j
29           └─ ←
30              ├─ ψ30
31              └─ call
32                 ├─ g
33                 └─ + (FPARENT=26, FSTHREAD=0)
34                    ├─ .i
35                    └─ .j
```

Figure C-4a.

```
23(1)   + (FPARENT = 16(1))
23(2)   ├── - (FPARENT = 16(2))
23(3)   │   ├── + (FPARENT = 16(3))
23(4)   │   │   ├── a
23(5)   │   │   └── * (FPARENT = 16(5))
23(6)   │   │       ├── - (FPARENT = 16(6))
23(7)   │   │       │   ├── .i
23(8)   │   │       │   └── 1
23(9)   │   │       └── 8
23(10)  │   └── 1
23(11)  └── .j
```

Figure C-4b.

```
30(1)   + (FPARENT = 16(1))
30(2)   ├── - (FPARENT = 16(2))
30(3)   │   ├── + (FPARENT = 16(3))
30(4)   │   │   ├── a
30(5)   │   │   └── * (FPARENT = 16(5))
30(6)   │   │       ├── - (FPARENT = 16(6))
30(7)   │   │       │   ├── .i
30(8)   │   │       │   └── 1
30(9)   │   │       └── 8
30(10)  │   └── 1
30(11)  └── .j
```

Figure C-4c.

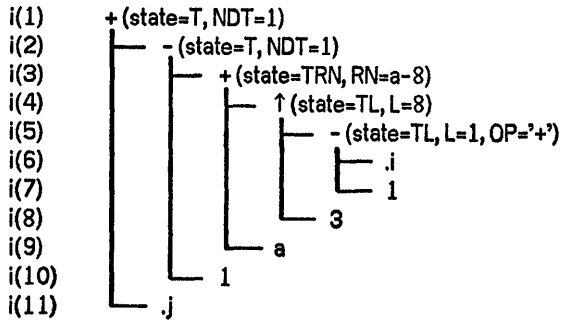


Figure C-4d.

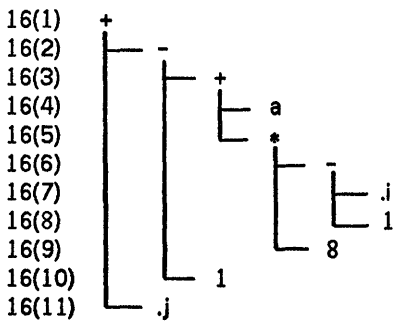


Figure C-5.

```
1   decr (lon=14, fon=11)
2     ├─ i
3     ├─ 8
4     ├─ 1
5     ├─ 1
6     └─ decr (lon=13, fon=10)
7          ├─ j
8          ├─ 8
9          ├─ 1
10         ├─ 1
11         └─ if (lon=12, fon=9)
12              ├─ eql
13              │   ├─ .i
14              │   └─ .j
15              ├─ ←
16              │   ├─ ψ16 (lon=5, fon=5, TN2)
17              │   └─ 0
18              └─ if
19                  ├─ gtr
20                  │   ├─ .i
21                  │   └─ .j
22                  ├─ ←
23                  │   ├─ ψ23 (lon=8, fon=8, TN2)
24                  │   └─ call (lon=7, fon=7, TN4)
25                  │       ├─ f
26                  │       └─ + (lon=6, fon=6, TN3)
27                  │           ├─ .i
28                  │           └─ .j
29                  └─ ←
30                      ├─ ψ30 (lon=11, fon=8, TN2)
31                      └─ call (lon=10, fon=7, TN5)
32                          ├─ g
33                          └─ + (lon=9, fon=6, TN3)
34                              ├─ .i
35                              └─ .j
```

Figure C-6.

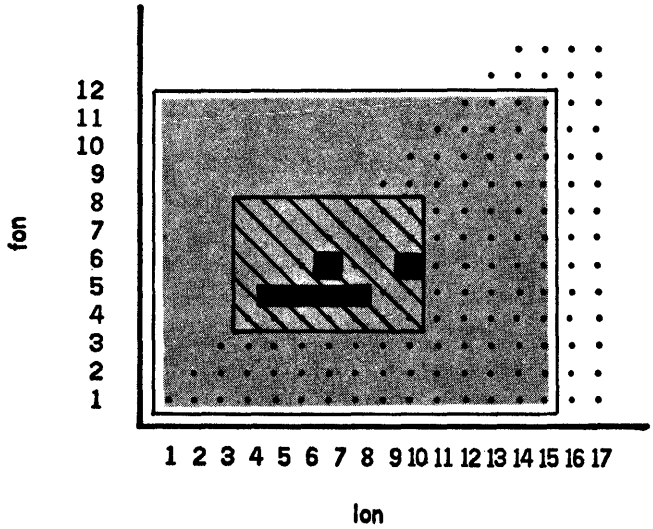


Figure C-7. lon/fon space for the example.

```
        MOV   #10,R1
L1:     MOV   R1,R5
        ASL   R5
        ASL   R5
        ASL   R5
        ADD   #A-9,R5
L3:     MOV   #10,R3
L5:     MOV   R5,R4
        ADD   R3,R4
        CMP   R3,R1
        BEQ   L6
        BR    L8
L6:     CLRB  @R4
        BR    L9
L8:     MOV   R1,R2
        ADD   R3,R2
        CMP   R1,R3
        BGT   L10
        BR    L13
L10:    MOV   R2,-(SP)
        JSR   PC,F
        MOVB  R0,@R4
        BR    L16
L13:    MOV   R2,-(SP)
        JSR   PC,G
        MOVB  R0,@R4
L16:    ADD   #2,SP
L9:     SUB   #1,R3
        BGT   L5
        SUB   #1,R3
        BGT   L1
```

Figure C-8. Code produced by CODE.

BIBLIOGRAPHY

[Bea72] Beatty, James C., "An Axiomatic Approach to Code Optimization for Expressions," JACM 19,4 (October 1972), 613-640.

[Flo63] Floyd, R. W., "Syntactic Analysis and Operator Precedence," JACM 10,3 (July 1963), 316-333.

[Fra70] Frailey, Dennis, "Expression Optimization Using Unary Complement Operators," Proceedings of the Symposium on Compiler Optimization, SIGPLAN 5,7 (July 1970).

[Hop69] Hopgood, F. R. A., Compiling Techniques, American Elsevier, New York, 1969.

[Ges72] Geschke, Charles M., "Global Program Optimizations," Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1972.

[Gri71] Gries, David, Compiler Construction for Digital Computers, John Wiley and Sons, New York, 1971.

[Low69] Lowery, E. S. and C. W. Medlock, "Object Code Optimization," CACM 12,1 (January 1969), 13-22.

[McK65] McKeeman, W., "Peephole Optimization," CACM 8,7 (July 1965), 443-444.

[Nak67] Nakata, Ikuo, "On Compiling Algorithms for Arithmetic Expressions," CACM 10,8 (August 1967), 492-94.

[Red69] Redziejowski, R. R., "On Arithmetic Expressions and Trees," CACM 12,2 (February 1969), 81-84.

[Set70] Sethi, R. and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," JACM 17,4 (October 1970), 715-728.

[Wul71] Wulf, W. A., D. B. Russell, and A. N. Habermann, "BLISS: A Language for Systems Programming," CACM 1,12 (December 1971), 780-790.

[Wul72a] Wulf, W. A., et al., BLISS-11 Programmer's Manual, Digital Equipment Corporation, Maynard, Mass., 1972.

[Wul72b] Wulf, W. A., "A Case Against the goto," Proceedings of the ACM National Conference, Boston, August 1972.

INDEX

A Complete Example 148-158
A Short Primer on Bliss 142-145
ABCOUNT 41-43
accessor 2
anycodesince 102

binding 71, 84, 88
binding of temporaries 9
binduses 74-75
bittest 95, 97
Bliss 2-3, 6-7, 9, 141-145
BNF 16
bound 85-87
branch-chaining 111-112, 118, 122
byte manipulation 92

CEILING 38-40
CHANGELIST 41-42
clearfield 92-93, 97
close 82, 85
CODE 8, 15, 89-108, 111, 113, 117, 130
code complexity 55
code generation 2, 8, 50, 89-90, 95, 101
code motion 11, 26-27, 29-30, 33, 35, 41, 43, 129
code size complexity 47
common subexpression 27, 48-49, 52-53, 58, 74-75, 91, 93, 113
common subexpression elimination 11
Conclusion 127-131
condition reversal 114
congruence 26, 31, 35, 37
congruent 26-27, 34-35, 37
constant folding 11, 19, 69
cost determination 71, 77
CRLEVEL 37-38
cross-jumping 110-113, 117, 131
CSE 26-27, 34-43, 48-55, 58, 64, 71, 75-76

CSPARENT 35
CSX 47

DEL 15-16
DELAY 6, 8, 15, 43, 45-69, 90-91, 103, 128-130
Delimiter 12, 15-17
desirable optimizations 45, 47-48, 129
dial 93, 97
dynamic temporaries 82, 85

effective address 63
empty 65, 85, 87, 110
epi-dominator 28-29
epilog 29, 31-32, 34, 41-43
essential order 22, 25
essential predecessor 41-42
evaluation order 6, 8-9, 22, 45-47, 50-52, 54-55, 57, 64, 68, 127, 129
EVO 47
execution order 90
execution order 8-9, 78, 89, 108
execution-order 72
EXPRESSION 17-19, 39
expression language 16, 25, 143

feasible 11, 27, 29, 35, 41-43, 45, 47-48, 86, 128-129
feasible global optimizations 6, 9, 11, 47
field isolation 92, 105-106
FINAL 6, 8, 107-125, 129, 131
fits 85-86
FLO 22, 34, 36, 43, 45, 48, 99-100, 131
FLOOR 38, 41
flow analysis 2, 6, 19, 21-22, 127
FLOWAN 19
fon 78-80, 130
fonfu 79, 84
fonlu 79, 84
formal intersection 31, 43
formal parent 34-35, 37
FPARENT 35
FSTHREAD 35, 37

generating data moves 91
genmove 102
gettn 75-77
GFEAS 11
goto 2, 38, 141, 145
GPOL 47
graph table 18, 36
GTHASH 35
GTHREAD 35

Hash Table 12, 14
HT 12, 14

independent 29
indexing 6, 63-65, 71, 78, 96, 116, 128
initial order 23-24, 38, 54
intersect 79, 85-86, 88
Introduction 1-9
isolate 93, 97

JM 37, 40

KFOLD 11

label assignment 71, 83
label cell 108, 110-112, 115-116
left recursion 16
LEVEL 38, 40
LEX 11, 15-16
lexeme 11-12, 15-20
lexical analysis 6, 11, 20
LEXSYNFLO 6, 8, 11-43
lifetime 9, 71, 78-79, 84-85, 128, 130
linear block 28-29, 31-32, 37-43
location 85-87, 102, 113
lon 78-80, 84, 130
lonfu 79
lonlu 79

MAKGT 18-19
masking 92, 105
MKLEVEL 37-39
MM 37-38, 40

Name Table 12, 14
NDT 58, 60
necessary constituent 42
NEG 58-60
NLOD 58-59
notecode 102
NT 12, 14

on-the-fly recognition 35, 38, 41-42
OP 58-59
open 82, 85, 87
operator grammar 15
OPNDPREF 47

PACK 8, 73, 82, 84-85, 87, 100
packanyreg 87
packanywhere 87
packmemory 87
packspecificreg 87
PD 37, 42
PDP-11 2, 47-48, 56, 60, 63, 69, 71, 78, 82, 84, 90-92, 103-104, 108, 111, 115-117, 119, 133, 135, 137
peephole 8, 107, 124
post-dominator 28-29
postlog 29, 31, 41-43
preference class 73, 75-76, 86, 100
preferencing 73, 82, 130
Primer on the PDP-11 134-140
pro-dominator 28-29
prolog 29, 31-32, 41-43

RANK 8, 84, 86-87
recursive descent 16-17, 38
redundant expression 27, 32, 35, 41, 129
redundant store 117
register allocation 8, 71

register use complexity 47, 55
reopen 85
RM 37-38, 40, 42
RUND 15
runtime stack 71, 80, 82, 111
RUX 47

shift 93, 97
shifting 92, 104-105
sif 15, 18, 20-21
sign preference 59-61
SP 81, 133
ST 12, 14
stack 17-19, 42
swhile 18, 20-21
switching 15-17
SYM 15, 18-19
symbol table 6, 8, 11-14, 41
SYN 11, 16
SYNTAX 15
syntax analysis 2, 6, 12, 15-16, 21, 69
syntax errors 20

target path 56, 58-59, 62, 64, 72, 76, 104
targeting 71-72, 82-83, 104, 127, 130
temporary 9, 32, 48, 56, 58-59, 64-65, 71, 76-78, 89-91, 100, 130
temporary name 71-79, 84-86, 100, 128
TLA 6, 8, 71-72, 74-75, 78-79, 82-83
tllist 75, 77
TN 71-73, 75-78, 84, 86, 88
TNBIND 15, 71-88, 90, 100, 128, 130
tnneeded 76-77
TPATH 58-59
translator 17-21
tryfit 86-87
trypermute 87-88

unary complement operator 56, 64
USELIST 41

VALIDITY 37

wantpref 75-77